Theoretische Informatik Zusammenfassung HS13

Gregor Wegberg

January 31, 2015

Contents

Todo list

Chapter 1

Alphabete, Wörter, Sprachen

Zur Darstellung von Daten werden Symbole verwendet. Diese wiederum werden zur Bildung von Wörtern genutzt. Und aus Wörtern bildet sich eine Sprache.

1.1 Alphabet

Definition 1.1.1. Eine endliche nichtleere Menge Σ heisst **Alphabet**. Die Elemente von Σ werden als **Buchstaben, Zeichen oder Symbole** bezeichnet.

Häufig benötigte Alphabete:

- $\Sigma_{\text{bool}} = \{0, 1\}$
- $\Sigma_{\text{lat}} = \{a, b, c, \ldots, z\}$
- $\Sigma_{\text{Tastatur}} = \Sigma_{\text{lat}} \cup \{A, B, C, \ldots, \_, <, >, \ldots\}$ (Alphabet aller Zeichen auf einer Tastatur)
- $\Sigma_m = \{0, 1, 2, \ldots, m-1\}$ (Alphabet für die m-adische Darstellung von Zahlen, $m \geq 1$)
- $\Sigma_{\text{logic}} = \{0, 1, x, (,), \wedge, \vee, \neg\}$ (Alphabet um Boole'sche Formeln darzustellen)

1.2 Wort

1.2.1 Grundlagen

Ein **Wort** wird aus Elementen eines zugrundeliegenden Alphabets gebildet.

Definition 1.2.1. Sei Σ ein Alphabet. Ein **Wort** über Σ ist eine endliche Folge von Buchstaben aus Σ.

Bemerkung 1.2.1. Σ^* ist die Menge aller Wörter über dem Alphabet Σ. $\Sigma^+ = \Sigma^* - \{\lambda\}$, also die Menge aller Wörter ohne das leere Wort.

Definition 1.2.2. Das **leere Wort** wird durch λ (oft auch ϵ) dargestellt und entspricht der leeren Folge.

Definition 1.2.3. Die **Länge** eines Wortes w, bezeichnet durch $|w|$, ist die Länge der Folge, d.h. die Anzahl vorkommender Buchstaben in w.

Bemerkung 1.2.2. Das leere Wort λ ist ein Wort über jedem Alphabet.

Definition 1.2.4. Sei $w \in \Sigma^*$ und $a \in \Sigma$. Dann ist $|w|_a$ definiert als die Anzahl der Vorkommen von a in w.

1.2.2 Beispiele für Kodierungen

1.2.2.1 Natürliche Zahlen

Sei m eine natürliche Zahl. Dann erzeugt die Funktion $\mathrm{Bin}(m) \in \Sigma^*_{\mathrm{bool}}$ die binäre Darstellung der natürlichen Zahl m. Damit $\mathrm{Bin}(m)$ eindeutig ist, soll die Funktion die kürzeste binäre Darstellung liefern (das erste Zeichen ist eine 1).

Die Umkehrfunktion ist $\mathrm{Nummer}(x) = \sum_{i=1}^{n} x_i \cdot 2^{n-i}$ und erzeugt für eine binäre Darstellung x die natürliche Zahl.

1.2.2.2 Graphen

Gerichtete Graphen $G = (V, E)$ (V ist die Knotenmenge, E die Kantenmenge) können durch eine Adjazenzmatrix M_G beschrieben werden: $M_G = [a_{ij}]$. Falls der Knoten $v_i \in V$ mit dem Knoten $v_j \in V$ verbunden ist, so gilt für M_G: $a_{ij} = 1$, sonst $a_{ij} = 0$. Die Adjazenzmatrix kann nun durch ein Wort über dem Alphabet $\Sigma = \{0, 1, \#\}$ beschrieben werden. Dazu schreibt man den Inhalt jeder Zeile nacheinander und trennt die Zeilen im entstehenden Wort durch #.

Möchte man einen gewichteten Graphen $G = (V, E, h)$ mit einer Funktion $h(e) \in \mathbb{N} - \{0\}$ für eine Kante $e \in E$, so kann dies ebenfalls über dem Alphabet $\Sigma = \{0, 1, \#\}$ gemacht werden. Wieder schreibt man den Inhalt jeder Zeile der Adjazenzmatrix nacheinander. Dabei kodiert man die Gewichtung mittels $\mathrm{Bin}(h(e))$, trennt die einzelnen Matrixeinträge durch # ab und Zeilen durch ##.

1.2.2.3 Bool'sche Formeln

Für Bool'sche Formeln verwenden wir das Alphabet $\Sigma_{\mathrm{logic}} = \{0, 1, x, (,), \wedge, \vee, \neg\}$. In Bool'schen Formeln kommen, im Gegensatz zu unserem Alphabet, beliebig viele Variabeln x_i vor. Unser Alphabet muss aber gleichzeitig endlich sein. Deshalb werden die Variabeln x_i durch das Wort $x\,\mathrm{Bin}(i)$ kodiert. Die restlichen Symbole können direkt übernommen werden.

1.2.3 Konkatenation

Definition 1.2.5. Die **Verkettung (Konkatenation)** für ein Alphabet Σ ist eine Abbildung $K : \Sigma^* \times \Sigma^* \to \Sigma^*$, so dass für alle $x, y \in \Sigma^*$

$$K(x, z) = x \cdot y = xy$$

gilt.

Bemerkung 1.2.3. Die Verkettung K über Σ ist eine assoziative Operation:

$$K(u, K(v, w)) = u \cdot (v \cdot w) = u \cdot (vw) = uvw = (u \cdot v) \cdot w = K(K(u, v), w)$$

Bemerkung 1.2.4. Für jedes $w \in \Sigma^*$ gilt

$$w \cdot \lambda = \lambda \cdot w = w$$

Bemerkung 1.2.5. Die Konkatenation ist nur für einelementige Alphabete kommutativ.

Bemerkung 1.2.6. Für alle $x, y \in \Sigma^*$ gilt:

$$|xy| = |x \cdot y| = |x| + |y|$$

Definition 1.2.6. Sei Σ ein Alphabet. Für alle $x \in \Sigma^*$ und alle $i \in \mathbb{N}$ wird die i-te Iteration x^i von x definiert als:

$$x^0 = \lambda, \quad x^1 = x, \quad x^i = x \cdot x^{i-1}$$

1.2.4 Teilworte

Definition 1.2.7. Seien $u, w \in \Sigma^*$ für ein Alphabet Σ.

- v ist ein **Teilwort** von $w \Leftrightarrow \exists x, y \in \Sigma^* : w = xvy$
- v ist ein **Suffix** von $w \Leftrightarrow \exists x \in \Sigma^* : w = xv$
- v ist ein **Präfix** von $w \Leftrightarrow \exists x \in \Sigma^* : w = vx$
- v ist ein **echtes** Teilwort/Suffix/Präfix von w genau dann, wenn $v \neq w$, $v \neq \lambda$ und v ist ein Teilwort/Suffix/Präfix von w

1.2.5 Ordnung

Definition 1.2.8. Sei $\Sigma = \{s_1, s_2, \ldots, s_m\}$ ein Alphabet für ein beliebiges $m \geq 1$. Weiter sei $s_1 < s_2 < s_3 < \ldots < s_m$ eine Ordnung auf Σ. Darauf basierend wird die **kanonische Ordnung** auf Σ^* für $u, v \in \Sigma^*$ wie folgt definiert:

$$\begin{aligned} u < v \Leftrightarrow \quad & |u| < |v| \\ & \vee (|u| = |v| \wedge u = x \cdot s_i \cdot u' \wedge v = x \cdot s_j \cdot v' \\ & \text{für beliebige } x, u', v' \in \Sigma^* \text{und } i < j) \end{aligned}$$

1.3 Sprache

Eine Sprache wird durch eine beliebige Menge von Wörtern über einem festen Alphabet gebildet.
Definition 1.3.1. Eine **Sprache** L über einem Alphabet Σ ist eine Teilmenge von Σ^*.

- $L_\emptyset = \emptyset$ ist die leere Sprache (hat keine Elemente)
- $L_\lambda = \{\lambda\}$ ist die einelementige Sprache, die nur das leere Wort enthält

Definition 1.3.2. Sind L_1 und L_2 zwei Sprachen über demselben Alphabet Σ, so ist

$$L_1 \cdot L_2 = L_1 L_2 = \{vw | v \in L_1, w \in L_2\}$$

die **Konkatenation** von L_1 und L_2.

Definition 1.3.3. Ist L eine Sprache über Σ, so wird definiert:

- $L^0 = L_\lambda$
- $L^{i+1} = L^i \cdot L, \quad \forall i \in \mathbb{N}$
- $L^* = \bigcup_{i \in \mathbb{N}} L^i$ (ist der **Kleene'sche Stern**)
- $L^+ = \bigcup_{i \in \mathbb{N} - \{0\}} L^i = L \cdot L^*$

Bemerkung 1.3.1.

- $\Sigma^i = \{w | w \in \Sigma^* \wedge |w| = i\}$
- $L_\emptyset L = L_\emptyset = \emptyset$
- $L_\lambda L = L$

Lemma 1.3.1. *Seien L_1, L_2, L_3 Sprachen über dem Alphabet Σ. Dann gilt $L_1 L_2 \cup L_1 L_3 = L_1(L_2 \cup L_3)$.*

Lemma 1.3.2. *Seien L_1, L_2, L_3 Sprachen über dem Alphabet Σ. Dann gilt $L_1(L_2 \cap L_3) \subseteq L_1 L_2 \cap L_1 L_3$*

Definition 1.3.4. Seien Σ_1, Σ_2 zwei Alphabete. Ein **Homomorphismus** von Σ_1^* nach Σ_2^* ist jede Funktion $h : \Sigma_1^* \to \Sigma_2^*$ mit folgenden Eigenschaften:

- $h(\lambda) = \lambda$
- $h(uv) = h(u) \cdot h(v) \quad \forall u, v \in \Sigma_1^*$

Bemerkung 1.3.2. Um einen Homomorphismus zu spezifizieren reicht es aus für alle Zeichen $a \in \Sigma_1$ $h(a)$ zu definieren.

1.4 Algorithmische Probleme

Ein Programm ist im Grunde eine Abbildung A, welche ein Wort über Σ_1 in ein Wort über Σ_2 abbildet: $A : \Sigma_1^* \to \Sigma_2^*$. Somit ist sowohl die Eingabe, wie auch die Ausgabe des Programms als Wort kodiert und A bestimmt für jedes Eingabewort ein bestimmtes Ausgabewort.

Zwei Programme A und B sind **äquivalent**, wenn für alle $x \in \Sigma_1^*$ $A(x) = B(x)$ gilt.

1.4.1 Entscheidungsproblem

Definition 1.4.1. Das **Entscheidungsproblem** (Σ, L) für ein gegebenes Alphabet Σ und eine gegebene Sprache $L \subseteq \Sigma^*$ ist, für jedes $x \in \Sigma^*$ zu entscheiden ob

$$x \in L \text{ oder } x \notin L.$$

Definition 1.4.2. Ein Algorithmus A **löst** das Entscheidungsproblem (Σ, L), falls für alle $x \in \Sigma^*$ gilt:

$$A(x) = \begin{cases} 1, & \text{falls } x \in L \\ 0, & \text{falls } x \notin L \end{cases}$$

In diesem Fall sagt man, dass A die Sprache L **erkennt**.

Definition 1.4.3. Wenn für eine Sprache L ein Algorithmus existiert, der L erkennt, so sagt man, dass L **rekursiv** ist.

Definition 1.4.4. Seien Σ, Γ zwei Alphabete. Wir sagen, dass ein Algorithmus A eine **Funktion** $f : \Sigma^* \to \Gamma^*$ berechnet, falls

$$\forall x \in \Sigma^* : A(x) = f(x)$$

Das Entscheidungsproblem ist ein Spezialfall einer Funktionsberechnung.

1.4.2 Relationsproblem

Definition 1.4.5. Seien Σ, Γ zwei Alphabete, und sei $R \subseteq \Sigma^* \times \Gamma^*$ eine Relation. Ein Algorithmus A löst das **Relationsproblem** R, falls für jedes $x \in \Sigma^*$ gilt:

$$(x, A(x)) \in R$$

1.4.3 Optimierungsproblem

Definition 1.4.6. Ein **Optimierungsproblem** ist ein 6-Tupel $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$:

- Σ_I ist das Eingabealphabet

- Σ_O ist das Ausgabealphabet
- $L \subseteq \Sigma_I^*$ ist die Sprache der zulässigen Eingaben
- $\mathcal{M}$ ist eine Funktion $\mathcal{M} : L \to \mathcal{P}(\Sigma_O^*)$. Für jedes $x \in L$ ist $\mathcal{M}(x)$ die Menge der zulässigen Lösungen für x
- cost ist eine Funktion cost : $\bigcup_{x \in L}(\mathcal{M} \times \{x\}) \to \mathbb{R}^+$ und ist die Preisfunktion
- goal $\in \{\text{Minimum}, \text{Maximum}\}$ ist das Optimierungsziel

Eine zulässige Lösung $\alpha \in \mathcal{M}(x)$ heisst **optimal** für den Problemfall x des Optimierungsproblems U, falls

$$\text{cost}(\alpha, x) = \text{Opt}_{\mathcal{U}} = \text{goal}\{\text{cost}(\beta, x) | \beta \in \mathcal{M}(x)\}$$

Definition 1.4.7. Ein Algorithmus A **löst** $\mathcal{U}$, falls für jedes $x \in L$:

1. $A(x) \in \mathcal{M}(x)$ ($A(x)$ ist eine zulässige Lösung des Problemfalls x von $\mathcal{U}$)
2. $\text{cost}(A(x), x) = \text{goal}\{\text{cost}(\beta, x) | \beta \in \mathcal{M}(x)\}$

Bemerkung 1.4.1. Oft wird die Spezifikation von Σ_I, Σ_O bei Optimierungsproblemen weggelassen. Man geht davon aus, dass die verwendeten Daten kodiert werden können für ein Σ_I, Σ_O. So bleiben noch vier Dinge übrig, die spezifiziert werden müssen:

1. die Menge der Problemfälle L, also die zulässigen Eingaben
2. die Menge der Einschränkungen, gegeben durch jeden Problemfall $x \in L$, und damit $\mathcal{M}(x)$ für jedes $x \in L$. $\mathcal{M}(x)$ gibt uns Lösungen, die den Einschränkungen genügen für ein gegebenes $x \in L$
3. die Kostenfunktion
4. das Optimierungsziel

1.4.3.1 Beispiel: Traveling Salesman Problem (TSP)

Eingabe: Ein gewichteter Graph (G, c), wobei $G = (V, E)$ ein Graph ist und $c : E \to \mathbb{N} - \{0\}$ die Kostenfunktion. Strikt formal müsste man das Eingabealphabet eingeben und mit diesem den Graphen kodieren.

Einschränkungen: Für jeden Problemfall (G, c) ist $\mathcal{M}(G, c)$ die Menge aller Hamiltonscher Kreise von G mit der Kostenfunktion e.

Kosten: Für jeden Hamiltonschen Kreis $H = v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1} \in \mathcal{M}(G, c)$:

$$\text{cost}((v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}), (G, c)) = \sum_{j=1}^{n} c\left(\{v_{i_j}, v_{i_{(j \mod n)+1}}\}\right)$$

Die Kosten jedes Hamiltonschen Kreises ist somit die Summe der Gewichte der besuchten Kanten.

Ziel: Minimum

Definition 1.4.8. Ein Optimierungsproblem $\mathcal{U}_1 = (\Sigma_I, \Sigma_O, L', \mathcal{M}, \text{cost}, \text{goal})$ ist ein **Teilproblem** vom Optimierungsproblem $\mathcal{U}_2 = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, falls $L' \subseteq L$.

1.4.3.2 Knotenüberdeckung

Definition 1.4.9. Eine **Knotenüberdeckung** eines Graphen $G = (V, E)$ ist jede Knotenmenge $U \subseteq V$, so dass jede Kante aus E mit mindestens einem Knoten aus U inzident ist. Eine Kante $\{u, v\} \in E$ ist inzident zu u und v.

Anders gesagt: Jede Kante muss an mindestens einem Ende in einem Knoten enden, der in der Knotenmenge der Knotenüberdeckung ist.

1.4.3.3 Beispiel: Maximale Clique Problem (MAX-CL)

Definition 1.4.10. Eine Clique eines Graphen $G = (V, E)$ ist jede Teilmenge $U \subseteq V$, so dass $\{\{u, v\} | u, v \in U, u \neq v\} \subseteq E$ (die Knoten von U bilden einen vollständigen Teilgraphen von G).

Das Maximale Clique Problem besteht nun darin eine Clique mit maximaler Kardinalität zu finden. Wir suchen also einen vollständigen Teilgraphen mit maximaler Anzahl von Knoten. Ein vollständiger Teilgraph ist ein Graph $H = (U, F), U \subseteq V, F \subseteq E$, so dass F alle Kanten aus E enthält, die zwei Knoten in U verbinden.

Eingabe: Ein (ungerichteter) Graph $G = (V, E)$

Einschränkung: $\mathcal{M}(G) = \{S \subseteq V | \{\{u, v\} | u, v \in S, u \neq v\} \subseteq E\}$

Kosten: Für jedes $S \in \mathcal{M}(G)$ ist $\text{cost}(S, G) = |S|$

Ziel: Maximum

1.4.3.4 Beispiel: Maximale Erfüllbarkeit (MAX-SAT)

Sei $X = \{x_1, x_2, \ldots\}$ die Menge der Bolle'schen Variabeln. Sei $Lit_X = \{x, \overline{x} | x \in X\}$ die Menge der Literale. Dabei ist $\overline{x}$ die negation von x. Eine Klausel ist eine beliebig grosse endliche Disjunktion von Literalen (z.B. $x_1 \vee \overline{x_2} \vee x_3$).

Eine Formel ist in **konjunktiver Normalform (KNF)**, falls sie eine Konjunktion von Klauseln ist. Also eine Konjunktion von Disjunktionen. Beispiel: $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_5})$.

Das Problem der maximalen Erfüllbarkeit ist, für eine gegebene Formel Φ in KNF, eine Belegung der Variabeln zu finden, die die maximale mögliche Anzahl an Klauseln von Φ erfüllt.

Eingabe: Eine Formel Φ in KNF

Einschränkung: Für jede Formel Φ über $\{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ ist $\mathcal{M}(\Phi) = \{0,1\}^n$. Jedes $\alpha = \alpha_1\alpha_2 \ldots \alpha_n \in \mathcal{M}(\Phi), \alpha_j \in \{0,1\}$ für $j = 1, 2, \ldots, n$ stellt eine Belegung für die Variable x_{i_j} dar.

Kosten: Für jedes Φ und jedes $\alpha \in \mathcal{M}(\Phi)$ ist $\text{cost}(\alpha, \Phi)$ die Anzahl der Klauseln, die durch α erfüllt werden.

Ziel: Maximum

1.4.3.5 Beispiel: Ganzzahlige Lineare Programmierung (integer linear programming, ILP)

Für ein gegebenes System von linearen Gleichungen und eine lineare Funktion von Unbekannten des linearen Systems soll eine Lösung dieses Systems berechnet werden. Die Lösung soll dabei minimal sein bezüglich der gegebenen linearen Funktion.

Eingabe: Eine $m \times n$ Matrix A und zwei Vektoren $b = (b_1, \ldots, b_m)^T$ und $c = (c_1, \ldots, c_n)$, wobei die Elemente von A, b, c ganze Zahlen sind.

Einschränkung: $\mathcal{M}(A, b, c) = \{X = (x_1, \ldots, x_n)^T \in \mathbb{N}^n | AX = b\}$. $\mathcal{M}(A, b, c)$ enthält somit alle Lösungsvektoren X für die $AX = b$ gilt.

Kosten: Für jedes $X = (x_1, \ldots, x_n) \in \mathcal{M}(A, b, c)$ ist $\text{cost}(X, (A, b, c)) = \sum_{i=1}^{n} c_i x_i$

Ziel: Minimum

1.4.4 Weitere Algorithmische Probleme

Definition 1.4.11. Sei Σ ein Alphabet, $x \in \Sigma^*$. Ein Algorithmus A **generiert** das Wort x, falls A für die Eingabe λ die Ausgabe x liefert.

Das folgende Programm erzeugt beispielsweise das Wort 1001:

```
begin
  write(1001);
end
```

Ein solches Programm kann als alternative Darstellung des Wortes verwendet werden.

Definition 1.4.12. Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine Sprache. Ein **Aufzählungsalgorithmus** A **für** L gibt für die Eingabe $n \in \mathbb{N} - \{0\}$ die Wortfolge $x_1, x_2, \ldots, x_n$ aus, wobei $x_1, x_2, \ldots, x_n$ die kanonisch ersten n Wörter aus L sind.

1.5 Kolmogorov-Komplexität

Definition 1.5.1. Für jedes Wort $x \in \Sigma^*_{\text{bool}}$ ist die **Kolmogorov-Komplexität** $K(x)$ des Wortes x die binäre Länge des kürzesten Pascal-Programms, das x generiert.

Lemma 1.5.1. *Es existiert eine Konstante d, so dass für jedes* $x \in \Sigma^*_{bool}$

$$K(x) \leq |x| + d$$

Die Länge eines solchen Programms ist also nicht wesentlich länger als das Wort selbst. Wir können für jedes Wort $x \in \Sigma^*_{\text{bool}}$ das Programm

```
begin
  write(x);
end
```

Man sieht, dass bloss das Wort x variabel im Programm ist und der Rest eine fixe Länge hat.

Das Wort x wird im Programm in binärer Form eingefügt und leistet deswegen zur binären Darstellung des Programms nur den Beitrag $|x|$.

Bemerkung 1.5.1. Für ein Wort $x \in \Sigma^*_{\text{bool}}$ braucht das Abspeichern von $\text{Bin}(|x|)$ so viel Platz: $\lceil \log_2(\text{Bin}(|x|) + 1) \rceil$

1.5.1 Beispiel #1

Um die Wörter der Form $y_n = 0^n \in \{0,1\}^*$ für jedes $n \in \mathbb{N} - \{0\}$ zu generieren, könnte das folgende Programm eingesetzt werden:

```
begin
  for l = 1 to n do
    write(0);
end
```

Wobei sich je nach Wort nur die Kodierung von n im Programm verändert. Somit erhalten wir für die Länge der Kodierung von n: $\lceil \log_2(n+1) \rceil$. Dies gibt uns eine Kolmogorov-Komplexität von $K(y_n) \leq \lceil \log_2(n+1) \rceil + c = \lceil \log_2 |y_n| \rceil + c$

1.5.2 Beispiel #2

Das Wort $z_n = 0^{n^2} \in \{0,1\}^*$ für jedes $n \in \mathbb{N} - \{0\}$ kann durch folgendes Programm dargestellt werden:

```
begin
  M := n;
  M := M × M;
  for l = 1 to M do
    write(0);
end
```

Hier ist wieder bloss n abhängig vom Wort, der Rest des Programms hat eine feste Länge. Wir erhalten somit

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + d \leq \lceil \log_2(\sqrt{|z_n|}) \rceil + d + 1$$

1.5.3 Weiteres

Definition 1.5.2. Die Kolmogorov-Komplexität einer natürlichen Zahl n ist $K(n) = K(\text{Bin(n)})$.
Lemma 1.5.2. *Für jede Zahl $n \in \mathbb{N} - \{0\}$ existiert ein Wort $w_n \in \Sigma^n_{bool}$, so dass $K(w_n) \geq |w_n| = n$, es existiert somit für jede Zahl n ein nichtkomprimierbares Wort der Länge n.*

Satz 1.5.1. Seien A und B zwei Programmiersprachen. Dann existiert die Konstante $c_{A,B}$, die nur von A und B abhängig ist, so dass:

$$|K_A(x) - K_B(x)| \leq c_{A,B} \quad \forall x \in \Sigma^*_{\text{bool}}$$

Daraus folgt, dass die verwendete Programmiersprache nicht relevant ist für die Berechnung der Kolmogorov-Komplexität.

Definition 1.5.3. Ein Wort $x \in \Sigma^*_{\text{bool}}$ ist **zufällig**, falls $K(x) \geq |x|$.

Eine Zahl $n \in \mathbb{N}$ ist zufällig, falls $K(n) = K(\text{Bin}(n)) \geq \lceil \log_2(n+1) \rceil - 1$.

Satz 2.2, Satz 2.3 (Primzahlsatz)

Chapter 2

Endliche Automaten

- Endliche Automaten sind das einfachste Berechnungsmodell, welches in der Informatik betrachtet wird.
- Sie entsprechen speziellen Programmen, die Entscheidungsprobleme lösen.
- Endliche Automaten verwenden dabei keine Variabeln.
- Die Eingabe wird nur einmal von links nach rechts gelesen.
- Nach dem Lesen des letzten Buchstabens steht das Resultat sofort fest.

2.1 Darstellung endlicher Automaten

Definition 2.1.1. Ein (deterministischer) **endlicher Automat (EA)** ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, F)$:

- Q ist eine endliche Menge von **Zuständen**
- Σ ist ein Alphabet, welches als **Eingabealphabet** bezeichnet wird
- $q_0 \in Q$ ist der **Anfangszustand**
- $F \subseteq Q$ ist die **Menge der akzeptierenden Zustände**
- δ eine Funktion $\delta : Q \times \Sigma \to Q$, welche als **Übergangsfunktion** bezeichnet wird. Daher bedeutet $\delta(q_i, a) = p$, dass falls M im Zustand $q_i \in Q$ den Buchstaben $a \in \Sigma$ liest, es in den Zustand $p \in Q$ übergeht.

Definition 2.1.2. Eine **Konfiguration** von M ist ein Element aus $Q \times \Sigma^*$. Falls M in der Konfiguration $(q_i, w) \in Q \times \Sigma^*$ ist, so bedeutet es, dass M aktuell im Zustand $q_i \in Q$ ist und noch den Suffix $w \in \Sigma^*$ des Eingabeworts zu lesen hat.

Die Konfiguration $(q_0, x) \in \{q_0\} \times \Sigma^*$ nennt man eine **Startkonfiguration** von M auf x. Die Berechnung des Worts x beginnt somit an der Startkonfiguration q_0.

Definition 2.1.3. Eine **Endkonfiguration** von M hat die Form $(q_i, \lambda) \in Q \times \{\lambda\}$

Definition 2.1.4. Ein **Schritt** von M ist eine Relation $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ und ist definiert durch

$$(q, w) \vdash_M (p, x) \Leftrightarrow w = ax,\ a \in \Sigma \wedge \delta(q, a) = p$$

Es handelt sich somit um eine Relation auf Konfigurationen des EA M. Es beschreibt den Übergang von einer Konfiguration in die nächste, nachdem der nächste Buchstabe von w (hier der Buchstabe a) gelesen wurde.

Definition 2.1.5. Eine **Berechnung** C von M ist eine endliche Folge $C = C_0, C_1, C_2, C_3, \ldots, C_n$ von Konfigurationen $(C_i \in (Q \times \Sigma^*),\ i = 0 \ldots n)$, so dass $C_i \vdash_M C_{i+1}$ gilt für alle $0 \leq i \leq n-1$.

Falls $C_0 = (q_0, x)$ und $C_n \in Q \times \{\lambda\}$, so ist C die Berechnung von M auf einer Eingabe $x \in \Sigma^*$.

Falls $C_n \in F \times \{\lambda\}$, so sagen wir, dass C eine **akzeptierende Berechnung** von M auf x ist, und dass M das Wort x akzeptiert.

Falls $C_n \in (Q - F) \times \{\lambda\}$, so ist C eine **verwerfende Berechnung** von M auf x, und dass M das Wort x verwirft.

Definition 2.1.6. Die von M akzeptierte Sprache $L(M)$ ist definiert als

$$L(M) := \{w \in \Sigma^* | \text{ die Berechnung von } M \text{ auf } w \text{ endet in einer Endkonfiguration } (q, \lambda) \text{ mit } q \in F\}$$

Definition 2.1.7. $\mathcal{L}(EA) = \{L(M) | M \text{ ist ein EA}\}$ ist die Klasse der Sprachen, die von endlichen Automaten akzeptiert werden. $\mathcal{L}(EA)$ bezeichnet man auch als die **Klasse der regulären Sprachen**, und jede Sprache L aus $\mathcal{L}(EA)$ als **regulär**.

Definition 2.1.8. Sei M ein EA. Wir definieren $\vdash_M^*$ als die reflexive und transitive Hülle der Schrittrelation $\vdash_M$ von M. Somit gilt

$$(q, w) \vdash_M^* (p, u) \Leftrightarrow (q = p \wedge w = u) \vee \exists k \in \mathbb{N} - \{0\} \text{ so dass}$$

- $w = a_1 a_2 a_3 a_4 \ldots a_k u,\ a_i \in \Sigma\ i = 1, \ldots, k$ und
- $\exists r_1, r_2, \ldots, r_{k-1} \in Q$, so dass $(q, w) \vdash_M (r_1, a_2 \ldots a_k u) \vdash_M (r_2, a_3 \ldots a_k u) \vdash_M \ldots \vdash_M (r_{k-1}, a_k u) \vdash_M (p, u)$

$(q, w) \vdash_M^* (p, u)$ sagt also aus, dass es eine Berechnung von M gibt, die ausgehen von der Konfiguration (q, w) zur Konfiguration (p, u) führt.

Definition 2.1.9. Sei $\hat{\delta} : Q \times \Sigma^* \to Q$ definiert durch:

- $\hat{\delta}(q, \lambda) = q$ für alle $q \in Q$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ für alle $a \in \Sigma,\ w \in \Sigma^*,\ q \in Q$

Wenn M im Zustand q ist und das Wort w zu lesen beginnt, dann bedeutet $\hat{\delta}(q, w) = p$ dass M im Zustand p enden wird. Oder anders gesagt, es gilt: $(q, w) \vdash_M^* (p, \lambda)$.

Gekürzt können wir nun $L(M)$ wie folgt definieren:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda),\ p \in F\} = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

2.2 Simulation

Lemma 2.2.1. *Sei Σ ein Alphabet und $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1), M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ zwei EA. Für $\circledcirc \in \{\cup, \cap, -\}$ existiert jeweils ein EA M mit $L(M) = L(M_1) \circledcirc L(M_2)$.*

Proof. Die Existenz von $L(M)$ basiert auf der Idee einer Konstruktion von M in welchem M_1 und M_2 simuliert werden. Dabei sind die Zustände von M Paare der Form $(q, p) \in Q_1 \times Q_2$.

Formale konstruktion von M: Sei $M = (Q, \Sigma, \delta, q_0, F_{\circledcirc})$ und:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_{01}, q_{02})$
- $\forall q \in Q_1, p \in Q_2, a \in \Sigma :\ \delta((q, p), a) = (\delta_1(q, a), \delta_2(p, a))$
- F ist:
  - Falls $\circledcirc = \cup$, dann ist $F = F_1 \times Q_2 \cup Q_1 \times F_2$
  - Falls $\circledcirc = \cap$, dann ist $F = F_1 \times F_2$
  - Falls $\circledcirc = -$, dann ist $F = F_1 \times (Q_2 - F_2)$

Um nun zu beweisen, dass ein solcher EA M für die Operation $\circledcirc$ existiert, reicht es zu zeigen dass folgende Gleichheit gilt: $\forall x \in \Sigma^* : \hat{\delta}((q_{01}, q_{02}), x) = (\hat{\delta}_1(q_{01}, x), \hat{\delta}_2(q_{02}, x))$. Der Beweis kann mittels Induktion über der Länge von x geführt werden. □

Diese Entwurfsmethode kann dazu eingesetzt werden endliche Automaten für komplexere Sprachen zu bauen, in dem EA für einfachere Sprachen zusammengesetzt werden. So kann die Sprache $L = \{x \in \Sigma_{\text{bool}}^* \mid |x|_0 \mod 3 = 1, |x|_1 \mod 3 = 2\}$ aus den Sprachen $L_1 = \{x \in \Sigma_{\text{bool}}^* \mid |x|_0 \mod 3 = 1\}$ und $L_2 = \{x \in \Sigma_{\text{bool}}^* \mid |x|_1 \mod 3 = 2\}$ konstruiert werden.

2.3 Beweise der Nichtexistenz

2.3.1 Lemma 3.3

Lemma 2.3.1 (Lemma 3.3)**.** *Sei $A = (Q, \Sigma, \delta_A, q_0, F)$ ein EA. Seien weiter $x, y \in \Sigma^*, x \neq y$, so dass $(q_0, x) \vdash_A^* (p, \lambda) \wedge (q_0, y) \vdash_A^* (p, \lambda)$ für ein $p \in Q$. Dann gilt für jedes $z \in \Sigma^* : xz \in L(A) \Leftrightarrow yz \in L(A)$.*

Anders ausgedrückt: Wenn wir zwei Wörter betrachten, die im EA A im selben Zustand landen, so haben wir ab diesem Moment keine Information mehr, welches der beiden Wörter wir gelesen haben. Konkatinieren wir nun ein Wort $z \in \Sigma^*$ zu den beiden ausgewählten Wörtern, so müssen beide Wörter in $L(A)$ sein oder beide nicht.

Dieses Lemma kann dazu eingesetzt werden zu zeigen, dass eine Sprache nicht regulär ist ($L \notin \mathcal{L}(EA)$). Dazu führt man den Beweis indirekt und nimmt an, dass L regulär sei und wendet das Lemma an. Dabei wird festgestellt, dass es ein z gibt, für welches $xz \in L$, aber $yz \notin L$, was ein Wiederspruch ist.

2.3.1.1 Beispiel

Es sei zu zeigen, dass $L = \{0^n 1^n | n \in \mathbb{N}\}$ nicht regulär ist.

Proof. Wir führen den Beweis indirekt mittels dem Lemma 3.3. Sei $L \in \mathcal{L}(EA)$. Somit existiert ein EA $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$.

Wir betrachten die Wörter $0^1, 0^2, \ldots, 0^{|Q|+1}$. Wir haben somit $|Q| + 1$ Wörter. Somit existieren $i, j \in \{1, 2, \ldots |Q| + 1\}, i < j$, so dass $\hat{\delta}(q_0, 0^i) = \hat{\delta}(q_0, 0^j)$. Nach Lemma 3.3 hat nun zu gelten: $\forall z \in \Sigma^* : 0^i z \in L \Leftrightarrow 0^j z \in L$. Dem ist aber nicht so. Für $z = 1^i$ erhalten wir: $0^i 1^i \in L$, aber auch $0^j 1^i \notin L$ $(i < j)$.

Somit ist $L \notin \mathcal{L}(EA)$. □

2.3.2 Pumping-Lemma

Lemma 2.3.2 (Pumping-Lemma für reguläre Sprachen)**.** *Sei L regulär. Dann existiert die Konstante $n_0 \in \mathbb{N}$, so dass jedes Wort $w \in \Sigma^*$ mit $|w| \geq n_0$ zerlegt werden kann in $w = yxz$, mit*

- *$|yx| \leq n_0$,*
- *$|x| \geq 1$ und*
- *entweder $\{yx^k z | k \in \mathbb{N}\} \subseteq L$ oder $\{yx^k z | k \in \mathbb{N}\} \cap L = \emptyset$.*

2.3.2.1 Beispiel

Es sei zu zeigen, dass $L = \{0^n 1^n | n \in \mathbb{N}\}$ nicht regulär ist.

Proof. Wir führen den Beweis indirekt mittels des Pumping-Lemmas. Sei $L \in \mathcal{L}(EA)$. Dann existiert eine Konstante n_0 mit den im Pumping-Lemma beschriebenen Eigenschaften. Wir betrachten das Wort $w = 0^{n_0}1^{n_0}$. Es gilt offensichtlich, dass $|w| \geq n_0$. Somit kann w zerlegt werden in $w = yxz$.

Bedingt durch die Eigenschaft $|yx| \leq n_0$ gilt hier, dass $y = 0^l$ und $x = 0^m$ für $l, m \in \mathbb{N}$. Bedingt durch die zweite Eigenschaft $|x| \geq 1$ ist $m \neq 0$. Nun prüfen wir ob mit diesen Eigenschaften, auch die dritte Pumping-Lemma Eigenschaft gilt. Wir erhalten also $\{yx^kz|z \in \mathbb{N}\} = \{0^{n_0-l}(0^m)^k1^{n_0}|k \in \mathbb{N}\} = \{0^{n_0-m+km}1^{n_0}|k \in \mathbb{N}\}$. Wählen wir $k = 0$, so erhalten wir das Wort $0^{n_0-m}1^{n_0}$, was nicht in L liegt, da $m \neq 0$. Für $k = 1$ hingegen liegt das Wort in L. Dies ist ein Wiederspruch. Somit ist L nicht regulär. □

2.3.3 Kolmogorov-Komplexität

... to add ...

2.4 Nichtdeterminismus

Determinismus bei einem deterministischen endlichen Automaten besagt, dass in jeder Konfiguration des EA eindeutig festgelegt ist, was im nächsten Schritt passiert. Somit bestimmt ein EA und ein Wort x eindeutig die Berechnung von x auf dem EA. Beim Nichtdeterminismus hingegen ist es erlaubt bei einer Konfiguration eine Auswahl an möglichen weiteren Schritten zu haben.
Definition 2.4.1. Ein **nichtdeterministischer endlicher Automat (NEA)** ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, F)$ mit

- Q eine endliche Menge von Zuständen
- Σ ist das Eingabealphabet
- $q_0 \in Q$ der Anfangszustand,
- $F \subseteq Q$ die Menge der akzeptierenden Zustände
- δ eine Funktion $\delta : Q \times \Sigma \to \mathcal{P}(Q)$. Die Übergangsfunktion.

Man bemerkt, dass Q, Σ, q_0, F die gleiche Bedeutungen haben bei einem EA wie auch bei einem NEA.

Der Unterschied liegt also in der δ-Funktion, welche als Wertebereich nicht Q, sondern $\mathcal{P}(Q)$ hat. Darin liegt auch der zentrale Unterschied. Ein Übergang kann in beliebig vielen Zuständen enden, oder keinem.

Ein NEA akzeptiert ein Wort, falls es eine Berechnung gibt, die in einem Zustand $p \in F$ landet. Es ist dabei egal, wo alle alternativen Berechnungen enden. Es reicht, wenn eine in einem akzeptierenden Zustand landet.

Definition 2.4.2. Zur Übergangsfunktion δ wird $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ wie folgt definiert:

- $\hat{\delta}(q, \lambda) = \{q\} \forall q \in Q$
- $\hat{\delta}(q, wa) = \{p|$ es existiert ein $r \in \hat{\delta}(q, w),$ so dass $p \in \delta(r, a)\} = \bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a)$

Aus der Definition sieht man, dass $\delta(q, w)$ jeweils die Menge aller Zustände aus Q liefert, die aus $q \in Q$ durch das vollständige Lesen von w erreichbar sind. Somit erhalten wir für einen NEA M: $L(M) = \{w \in \Sigma^* | \ \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$.

Satz 2.4.1. Zu jedem NEA M existiert ein EA A, so dass $L(M) = L(A)$.

Proof. Sei $M = (Q, \Sigma, \delta_M, q_0, F)$ ein NEA und $A = (Q_A, \Sigma_A, \delta_A, q_{0A}, F_A)$ ein EA. A können wir dabei aus M wie folgt konstruieren:

- $Q_A = \{\langle P \rangle \mid P \subset Q\}$
- $\Sigma_A = \Sigma$
- $q_{0A} = \langle \{q_0\} \rangle$
- $F_A = \{\langle P \rangle \mid P \subseteq Q, P \cap F \neq \emptyset\}$
- $\delta_A : Q_A \times \Sigma_A \to Q_A$ und definiert wie folgt:

$$\forall \langle P \rangle \in Q_A, \forall a \in \Sigma_A : \delta_A(\langle P \rangle, a) = \left\langle \bigcup_{p \in P} \delta_M(p, a) \right\rangle$$

□

Chapter 3

Turingmaschinen

3.1 Turingmaschine

Definition 3.1.1. Eine Turingmaschine (TM) ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$:

- Q ist eine endliche Menge von **Zuständen** und wird als **Zustandsmenge von** M bezeichnet.
- Σ ist das **Eingabealphabet**, wobei die Symbole ¢, $\sqcup$ nicht in Σ sind.
- Γ ist das **Arbeitsaplhabet**, wobei
  - $\Sigma \subseteq \Gamma$
  - ¢, $\sqcup \in \Gamma$
  - $\Gamma \cap Q = \emptyset$

  Somit enthält das Arbeitsalphabet die Symbole, die auf dem Arbeitsband vorkommen dürfen. Dazu gehört die Eingabe, da diese am Anfang auf dem Band steht.
- $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}) \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$ ist die **Übergangsfunktion von** M. Es gilt jeweils

  $$\forall q \in Q : \ \delta(q, ¢) \in Q \times \{¢\} \times \{R, N\}$$

  Diese spezielle Eigenschaft beim Lesen des Symbols ¢ verbietet es das Symbol zu ersetzen und den Lesekopf nach Links zu bewegen. Dies macht Sinn, da wir bereits am linken Rand angelangt sind und das Band dort erst beginnt.
  Das Resultat $\delta(p, Y) = (q, X, Z) \in Q \times \Gamma \times \{L, R, N\}$ sagt aus, dass beim Lesen des Symbols Y im Zustand p wird folgendes passieren: Das gelesene Symbol Y wird durch X ersetzt und der Lesekopf in Richtung Z bewegt. Der neue Zustand ist q.
- $q_0 \in Q$ ist der **Anfangszustand**

- $q_{\text{accept}} \in Q$ ist der **akzeptierende Zustand**. Im Unterschied zu einem EA, hat eine TM genau einen akzeptierenden Zustand. Ist dieser Erreicht, werden keine Aktionen mehr ausgeführt auf der TM.
- $q_{\text{reject}} \in Q - \{q_{\text{accept}}\}$ ist der **verwerfende Zustand**. Wenn die TM diesen Zustand erreicht, so verwirft die TM die Eingabe. Daraus folgt, dass die TM nicht die gesamte Eingabe lesen muss, um zu akzeptieren/zu verwerfen.

Definition 3.1.2. Eine **Konfiguration** C einer TM M ist ein Element aus

$$\text{Konf}(M) = \{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^+ \cup Q \cdot \{¢\} \cdot \Gamma^+$$

Eine Konfiguration $w_1 q a w_2 \sqcup \sqcup \ldots$ mit $w_1 \in \{¢\}\Gamma^*, w_2 \in \Gamma^*, a \in \Gamma, q \in Q$ beschreibt die folgende Situation:

- Die TM M befindet sich im Zustand q.
- Der Inhalt des Bandes entspricht $w_1 a w_2 \sqcup \sqcup \ldots$
- Der Kopf ist gerade beim $|w_1|$+1-ten Symbol auf dem Band. Liest/Schreibt gerade a.

Eine Konfiguration der Form $p¢w \sqcup \sqcup \ldots$ sagt aus, dass wir im Zustand p sind und das auf dem Band $¢w \sqcup \sqcup \ldots$ steht.

Definition 3.1.3. Die **Startkonfiguration** für ein Eingabewort $x \in \Sigma^*$ ist: $q_0 ¢ x$.

Definition 3.1.4. Ein **Schritt** ist eine Relation $\vdash_M \subseteq \text{Konf}(M) \times \text{Konf}(M)$. Wobei das erste Element des Tupels eine Konfiguration ist und das zweite Element wieder eine Konfiguration. Vergleicht man beide Konfigurationen, so ist ersichtlich, was die Übergangsfunktion war.

Definition 3.1.5. Eine **Berechnung** ist eine potentiell unendliche Folge von Konfigurationen: $C_0, C_1, C_2, \ldots$, so dass $C_i \vdash_M C_{i+1}$ $i = 0, 1, 2, \ldots$. Wenn $C_0 \vdash_M C_1 \vdash_M \cdots \vdash_M C_i$ für ein $i \in \mathbb{N}$, dann $C_0 \vdash_M^* C_i$.

Definition 3.1.6. Eine **Berechnung** von M auf x beginnt mit einer Startkonfiguration $C_0 = q_0 ¢ x$ und läuft entweder unendlich lange, oder endet in einer Konfiguration $C_i = w_1 q w_2$ wobei $q \in \{q_{\text{accept}}, q_{\text{reject}}\}$.

Definition 3.1.7.
- Die Berechnung von M auf x heisst **akzeptierend**, falls sie in einer Konfiguration $C = w_1 q_{\text{accept}} w_2$ endet.
- Die Berechnung ist **verwerfend**, wenn sie in einer Konfiguration $C = w_1 q_{\text{reject}} w_2$.
- Eine **nicht-akzeptierende** Berechnung ist gegeben, wenn sie verworfen wird oder die Berechnung unendlich ist.

Definition 3.1.8. Die von einer Turingmaschine M akzeptierte Sprache ist

$$L(M) = \{w \in \Sigma^* \mid q_0 ¢ w \vdash_M^* y q_{\text{accept}} z \wedge y, z \in \Gamma^*\}$$

Definition 3.1.9. M **berechnet** eine Funktion $F : \Sigma^* \to \Gamma^*$, falls

$$\forall x \in \Sigma^* : q_0 ¢ x \vdash_M^* q_{\text{accept}} ¢ F(x)$$

Definition 3.1.10. Eine Sprache L heisst **rekursiv aufzählbar**, falls eine TM M existiert, so dass $L = L(M)$.

Die Menge aller rekursiv aufzählbaren Sprachen entspricht somit:

$$\mathcal{L}_{RE} = \{L(M) \mid M \text{ ist eine TM}\}$$

Definition 3.1.11. Eine Sprache $L \subseteq \Sigma^*$ heisst **rekursiv (entscheidbar)**, falls $L = L(M)$ für eine TM M, wobei für alle $x \in \Sigma^*$ gilt:

- $q_0 ¢ x \vdash_M^* y q_{\text{accept}} z,\ y, z \in \Gamma^*$, falls $x \in L$
- $q_0 ¢ x \vdash_M^* y q_{\text{reject}} z,\ y, z \in \Gamma^*$, falls $x \notin L$

Wenn eine TM also immer im akzeptierenden oder verwerfenden Zustand landet, also nie unendlich lange läuft für eine beliebige Eingabe, so sagt man, dass die **TM immer hält**.

Definition 3.1.12. Eine TM, die immer hält, ist ein formales Modell des Begriffs Algorithmus.

Definition 3.1.13. Die Menge der **rekursiven (algorithmisch erkennbaren) Sprachen** ist:

$$\mathcal{L}_R = \{L(M) \mid M \text{ ist eine TM, die immer hält}\}$$

3.2 Mehrband-Turingmaschinen

Für eine positive ganze Zahl k hat eine k-Band-Turingmaschine die folgenden Komponenten

- eine endliche Kontrolle (das Programm)
- ein endliches Band mit einem *Lese*kopf
- k Arbeitsbänder, jedes mit einem Lese/Schreibkopf

dabei haben wir folgende Eigenschaften am Anfang:

- Auf dem Eingabeband befindet sich $¢w\$$. ¢ markiert den Anfang des Bandes (ganz Links) und \$ das Ende des Bandes (ganz Rechts).
- Der Lesekopf des Eingabebandes beginnt bei ¢.
- Der Inhalt der k Arbeitsbänder ist $¢ \sqcup \sqcup \ldots$ und der Lese/Schreibkopf beginnt bei ¢.
- Die endliche Kontrolle befindet sich im Anfangszustand q_0.

Alle $k+1$ Köpfe dürfen sich nach Links und Rechts bewegen. Falls sie auf ein ¢ stossen, so dürfen sie nicht mehr weiter nach Links lesen. Beim Antreffen von \$ auf dem Eingabeband, darf der Lesekopf nicht weiter nach Rechts. Der Lesekopf auf dem Eingabeband darf nicht schreiben. Somit bleibt der Inhalt des Eingabebandes während der gesamten Berechnung unverändert.

Die Felder der Bänder können von links nach rechts nummeriert werden. Dabei ist jeweils ¢ das 0-te Symbol des Bandes.

Eine Konfiguration einer k-Band-Turingmaschine M kann durch ein Tupel der Form

$$(q, w, i, u_1, i_2, u_2, i_2, \ldots, u_k, i_k) \in Q \times \Sigma^* \times \mathbb{N} \times (\Gamma \times \mathbb{N})^k$$

was folgende Bedeutung hat:

- M befindet sich im Zustand q
- der Inhalt des Eingabebandes ist ¢w\$ und der Lesekopf des Eingabebandes zeigt auf das i-te Feld des Eingabebandes
- für $j \in \{1, 2, \ldots, k\}$ ist der Inhalt des j-ten Bandes ¢$u_j \sqcup\sqcup \ldots$ und $i_j \leq |u_j|$ ist die Position des j-ten Lese/Schreibkopfs.

Definition 3.2.1. Eine Maschine A ist äquivalent zu einer Maschine B, falls für jede Eingabe $x \in \Sigma^*_{\text{bool}}$:

- A akzeptiert $x \Leftrightarrow B$ akzeptiert x
- A verwirft $x \Leftrightarrow B$ verwirft x
- A arbeitet unendlich lange auf $x \Leftrightarrow B$ arbeitet unendlich lange auf x

Lemma 3.2.1. *Für jede Mehrband-TM (MTM) existiert eine äquivalente TM.*

3.3 Nichtdeterministische Turingmaschinen

... nicht wirklich relevant derzeit ...

3.4 Kodierung von Turingmaschinen

Definition 3.4.1. $\text{Kod}(M)$ kodiert eine Turingmaschine M über Σ_{bool}.

Definition 3.4.2. Die Menge aller Kodierungen aller Turingmaschinen ist

$$\text{KodTM} = \{\text{Kod}(M) \mid M \text{ ist eine TM}\}$$

Definition 3.4.3. Sei $x \in \Sigma^*_{\text{bool}}$. Für jedes $i \in \mathbb{N} - \{0\}$ kodiert x die i-te TM, falls

- $x = \text{Kod}(M)$ für eine TM M, und
- die Menge $\{y \in \Sigma^*_{\text{bool}} \mid y \text{ ist vor } x \text{ in kanonischer Ordnung}\}$ enthält genau $i-1$ Wörter, die Kodierungen von Turingmaschinen sind.

Chapter 4

Berechenbarkeit

4.1 Die Methode der Diagonalisierung

Definition 4.1.1. Seien A, B zwei Mengen.

- $|A| \leq |B|$, falls eine injektive Funktion $f : A \to B$ existiert.
- $|A| = |B|$, falls eine eine Bijektion $f : A \to B$ bzw. $f' : B \to A$ existiert. Dies ist gleichbedeutend mit der Anforderung, dass $|A| \leq |B| \wedge |B| \leq |A|$ ist.
- $|A| < |B|$, falls $|A| \leq |B|$ und es existiert keine injektive Abbildung $f : B \to A$.

Definition 4.1.2. Eine Menge A ist **abzählbar**, falls A endlich ist oder $|A| = |\mathbb{N}|$.

Anders definiert: A ist abzählbar $\Leftrightarrow$ es existiert eine injektive Funktion $f : A \to \mathbb{N}$.

Intuitive bedeutet die Abzählbarkeit, dass die Elemente der abzählbaren Menge nummeriert werden können. Sie führt also eine lineare Ordnung ein.
Lemma 4.1.1. *Sei Σ ein beliebiges Alphabet. Dann ist Σ^* abzählbar.*

Satz 4.1.1. Die Menge der Turingmaschinenkodierungen KodTM ist abzählbar.

Lemma 4.1.2. $|\mathbb{Q}^+| = |\mathbb{N}|$. *Somit ist die Menge der postiven rationalen Zahlen abzählbar.*

Lemma 4.1.3. $(\mathbb{N} - \{0\}) \times (\mathbb{N} - \{0\})$ *ist abzählbar.*

Satz 4.1.2. $[0, 1] \subseteq \mathbb{R}$ ist nicht abzählbar.

Satz 4.1.3. $\mathcal{P}(\Sigma^*_{\text{bool}})$ ist nicht abzählbar.

Korollar 4.1.1. $|\,\mathrm{KodTM}\,| < |\mathcal{P}(\Sigma^*_{bool})|$ *und somit existieren unendlich viele nicht rekursiv abzählbare Sprachen über* Σ_{bool}.

4.1.1 Diagonalsprache

Sei w_i das i-te Wort in kanonischer Ordnung über Σ_{bool}. Sei M_i die i-te Touringmaschine. Daraus definieren wir die Matrix $A = [d_{ij}]_{i,j=1,\ldots,\infty}$, wobei $d_{ij} = 1 \Leftrightarrow M_i$ akzeptiert w_j.

Somit bestimmt die i-te Zeile der Matrix A die Sprache $L(M_i) = \{w_j|\ d_{ij} = 1 \text{ für alle } j \in \mathbb{N} - \{0\}\}$.

Nun lässt sich durch die Idee der Diagonalisierung eine Sprache erzeugen, die keiner der Sprachen $L(M_i)$ entspricht.

$$\begin{aligned} L_{\mathrm{diag}} &= \{w \in \Sigma^*_{\mathrm{bool}} | w = w_i \text{ für ein } i \in \mathbb{N} - \{0\} \wedge M_i \text{ akzeptiert } w_i \text{ nicht}\} \\ &= \{w \in \Sigma^*_{\mathrm{bool}} | w = w_i \text{ für ein } i \in \mathbb{N} - \{0\} \wedge d_{ii} = 0\} \end{aligned}$$

Satz 4.1.4. $L_{\mathrm{diag}} \notin \mathcal{L}_{\mathrm{RE}}$

4.2 Die Methode der Reduktion

Definition 4.2.1. Seien $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ zwei Sprachen. L_1 **ist auf** L_2 **rekursiv reduzierbar**, $L_1 \leq_R L_2$, falls

$$L_2 \in \mathcal{L}_R \Rightarrow L_1 \in \mathcal{L}_R$$

Die Intuition hinter der Bezeichnung $L_1 \leq_R L_2$ ist, dass L_2 bezüglich der algorithmischen Lösbarkeit mindestens so schwer wie L_1 ist.

Wäre also L_2 algorithmisch lösbar, sprich es gibt eine TM M mit $L(M) = L_2$, dann wäre auch L_1 durch eine TM M' lösbar. Die TM M' könnte mit Hilfe von M konstruiert werden.

4.2.1 EE-Reduktion

Das Ziel ist es nun für zwei Sprachen zu zeigen, dass $L_1 \leq_R L_2$ gilt. Dazu sucht man eine TM M, die für jede Eingabe x und das Entscheidungsproblem (Σ_1, L_1) eine Ausgabe y generiert. Diese Ausgabe wiederum ist die Eingabe für das Entscheidungsproblem (Σ_2, L_2). Dieses Umschreiben soll dabei den Effekt haben, dass die Lösung des Entscheidungsproblems (Σ_2, L_2) auf y gerade der Lösung des Entscheidungsproblems (Σ_1, L_1) auf x entspricht.

Definition 4.2.2. Seien $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ zwei Sprachen. L_1 ist auf L_2 **EE-reduzierbar**, $L_1 \leq_{EE} L_2$, wenn eine TM M existiert, die eine Abbildung $f_M : \Sigma_1^* \to \Sigma_2^*$ für alle $x \in \Sigma_1^*$ berechnet mit der Eigenschaft, dass

$$x \in L_1 \Leftrightarrow f_M(x) \in L_2.$$

In einem solchen Fall spricht man davon, dass die TM M die Sprache L_1 auf die Sprache L_2 reduziert.

Lemma 4.2.1. *Seien $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ zwei Sprachen. Falls $L_1 \leq_{EE} L_2$, dann auch $L_1 \leq_R L_2$.*

Lemma 4.2.2. *Sei Σ ein Alphabet. Für jede Sprache $L \subseteq \Sigma^*$ gilt:*

$$L \leq_R L^C \wedge L^C \leq_R L$$

Korollar 4.2.1. $L_{diag}^C \notin \mathcal{L}_R$

Lemma 4.2.3. $L_{diag}^C \in \mathcal{L}_{RE}$

Korollar 4.2.2. $L_{diag}^C \in \mathcal{L}_{RE} - \mathcal{L}_R \Rightarrow \mathcal{L}_R \subsetneq \mathcal{L}_{RE}$

4.2.1.1 Rekursiv aufzählbare Sprachen, die nicht rekursiv sind

Definition 4.2.3. Die **universelle Sprache** ist definiert durch

$$L_U = \{\text{Kod}(M)\#w \mid w \in \Sigma_{\text{bool}}^*, w \in L(M)\}$$

Satz 4.2.1. Es existiert die **universelle** TM U, so dass $L(U) = L_U$. Daraus folgt, dass $L_U \in \mathcal{L}_{RE}$.

Satz 4.2.2. $L_U \notin \mathcal{L}_R$

Proof. Es reicht zu zeigen, dass $L_{\text{diag}}^C \leq_R L_U$ gilt. Dies sagt aus, dass L_U mindestens so schwer ist wie L_{diag}^C. Wir wissen bereits, dass $L_{\text{diag}}^C \notin \mathcal{L}_R$ und daraus würde automatisch folgen, dass $L_U \notin \mathcal{L}_R$.

Sei A ein Algorithmus, der L_U entscheidet. Damit bauen wir einen Algorithmus B, der A verwendet, um die Sprache L_{diag}^C zu entscheiden. B nimmt als Eingabe $x \in \Sigma_{\text{bool}}^*$ entgegen. Diese Eingabe wird zuerst an das Unterprogramm C weitergeleitet. C berechnet an welcher Stelle i das Wort $x = w_i$ in kanonischer Reihenfolge steht. Mittels dieser Information erzeugt es zusätzlich $\text{Kod}(M_i)$, also die i-te TM. Das Resultat von C ist somit $\text{Kod}(M_i)\#w_i$. Diese Ausgabe wird als Eingabe an A geleitet. Wir wissen, dass A entscheidet, ob M_i auf w_i hält. Nach der Vorgabe hält A auf der gegebenen Eingabe. Daraus folgt, dass auch B hält, da sowohl C als auch A Algorithmen sind. Weiter ist offensichtlich, dass $L(B) = L_{\text{diag}}^C$. Somit ist L_U mindestens so schwer wie L_{diag}^C. Daraus folgt, dass $L_U \notin \mathcal{L}_R$. □

Beweis mittels EE-Reduktion. Es gilt nach unserem bisherigen Wissensstand: $L_{\text{diag}}^C \leq_{EE} L_U \Rightarrow L_{\text{diag}}^C \leq_R L_U$. Somit zeigen wir nun den Beweis mittels EE-Reduktion. Sei M eine TM, die eine Abbildung $f_M : \Sigma_{\text{bool}} \to \{0, 1, \#\}^*$ berechnet, so dass

$$x \in L_{\text{diag}}^C \Leftrightarrow f_M(x) \in L_U.$$

M arbeitet nun wie folgt. Für eine Eingabe x berechnet M zuerst ein i, so dass $x = w_i$ (x ist das i-te Wort in kanonischer Reihenfolge). Danach berechnet M die Kodierung $\text{Kod}(M_i)$ der i-ten TM. M hält mit dem Inhalt $\text{Kod}(M_i)\#w_i$ auf dem Band. Weil $x = w_i$, ist es nach Definition von L^C_{diag} offensichtlich, dass

$$\begin{aligned} x = w_i \in L^C_{\text{diag}} &\Leftrightarrow M_i \text{ akzeptiert } w_i \\ &\Leftrightarrow w_i \in L(M_i) \\ &\Leftrightarrow \text{Kod}(M_i)\#w_i \in L_U \end{aligned}$$

□

4.2.1.2 Halteproblem

Definition 4.2.4. Das **Halteproblem** ist das Entscheidungsproblem $(\{0, 1, \#\}^*, L_H)$, wobei

$$L_H = \{\text{Kod}(M)\#x \mid x \in \Sigma^*_{\text{bool}} \wedge M \text{ hält auf } x\}$$

Satz 4.2.3. $L_H \notin \mathcal{L}_R$

Beweis auf der Ebene von Programmen. Wir möchten zeigen, dass $L_U \leq_R L_H$ gilt. Wir zeigen also, dass L_H mindestens so schwierig ist wie L_U.

Sei $L_H \in \mathcal{L}_R$. Daher existiert ein Algorithmus H, der L_H akzeptiert. Wir bauen nun damit den Algorithmus U wie folgt. Als Eingabe erhalten wir das Wort w. Ein Unterprogramm C prüft, ob $w = x\#y$ mit $x = \text{Kod}(M)$ für eine TM M ist. Falls nicht, so verwirft C die Eingabe, was zum Verwerfen der Eingabe in U folgt.

Falls die Eingabe das gewünschte Format hat, wird diese weiter an unser Unterprogramm H geleitet. H prüft nun, ob die TM M auf der Eingabe y hält. Falls nicht, so wissen wir, dass $y \notin L(M) \Rightarrow x\#y \notin L_U$ und verwerfen die Eingabe in U. Hält hingegen M, so wissen wir, dass M nach **endlich** vielen Schritten y akzeptiert hat oder nicht. Somit übergeben wir die Eingabe einem weiteren Unterprogramm S, welches M auf der Eingabe y simuliert. Falls M in q_{accept} landet, so gilt $x\#y \in L_U$, sonst nicht.

Es ist offensichtlich, dass $L(U) = L_U$. Daraus folgt: $L_U \leq_R L_H \wedge L(U) \notin \mathcal{L}_R \Rightarrow L_H \notin \mathcal{L}_R$

□

4.2.1.3 Leere Sprache

Definition 4.2.5.

$$L_{\text{empty}} = \{\text{Kod}(M) \mid L(M) = \emptyset\}$$

Enthält die Kodierung aller TM, die die leere Menge akzeptieren.

Korollar 4.2.3.

$L^C_{empty} = \{x \in \Sigma^*_{bool} \mid (x \notin \text{Kod}(M') \textit{ für alle TM } M') \vee (x = \text{Kod}(M) \wedge L(M) \notin \emptyset)\}$

Lemma 4.2.4. $L^C_{empty} \in \mathcal{L}_{RE}$

Lemma 4.2.5. $L^C_{empty} \notin \mathcal{L}_R$

Korollar 4.2.4. $L_{empty} \notin \mathcal{L}_R$

Korollar 4.2.5. *Die Sprache* $L_{EQ} = \{\text{Kod}(M)\#\text{Kod}(M') \mid L(M) = L(M')\}$ *ist nicht entscheidbar* ($L_{EQ} \notin \mathcal{L}_R$).

4.2.2 Satz von Rice

Definition 4.2.6. Eine Sprache $L \subseteq \{\text{Kod}(M) \mid M \text{ eine TM}\}$ heisst **semantisch nichttriviales Entscheidungsproblem über Turingmaschinen**, falls die folgenden Bedingungen erfüllt sind:

- Es gibt eine TM M_1, so dass $\text{Kod}(M_1) \in L$. Es gilt somit $L \neq \emptyset$
- Es gibt eine TM M_2, so dass $\text{Kod}(M_2) \notin L$. L enthält also nicht die Kodierung aller Turingmaschinen
- Für zwei TM A, B gilt: $L(A) = L(B) \Rightarrow (\text{Kod}(A) \in L \Leftrightarrow \text{Kod}(B) \in L)$

Lemma 4.2.6. $L_{H,\lambda} = \{\text{Kod}(M) \mid M \textit{ hält auf } \lambda\}$. $L_{H,\lambda}$ *ist ein Spezialfall des Halteproblems, womit wir* $L_{H,\lambda} \notin \mathcal{L}_R$ *erhalten.*

Satz von Rice relevant?

Das Post'sche Korrespondenzproblem ausgelassen

Kapitel 5.6, Seite 199, ausgelassen

Chapter 5

Komplexitätstheorie

In der Komplexitätstheorie wird primär mit Mehrband-Turingmaschinen gearbeitet.

5.1 Komplexitätsmasse

Definition 5.1.1. Sei M eine MTM, die immer hält. Sei weiter Σ das Eingabealphabet von M. Sei $D = C_1, C_2, \ldots, C_l$ die Berechnung von M auf einem $x \in \Sigma^*$. Dann ist die **Zeitkomplexität** $\mathrm{Time}_M(x)$ definiert durch

$$\mathrm{Time}_M(x) = k - 1.$$

Also der Anzahl Berechnungsschritte, die M auf x durchläuft ($|D|$).

Definition 5.1.2. Die **Zeitkomplexität von** M ist die Funktion $\mathrm{Time}_M : \mathbb{N} \to \mathbb{N}$ definiert durch

$$\mathrm{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid x \in \Sigma^n\}, \quad n \in \mathbb{N}.$$

Definition 5.1.3. Sei $k \in \mathbb{N} - \{0\}$ und M eine k-Band-Turingmaschine, die immer hält. Dann ist

$$C = (q, x, i, \alpha_1, i_1, \alpha_2, i_2, \ldots, \alpha_k, i_k)$$

eine Konfiguration von M. Die **Speicherplatzkomplexität einer Konfiguration** C ist dann

$$\mathrm{Space}_M(C) = \max\{|\alpha_i| \mid i = 1, \ldots, k\}.$$

Somit ist die Speicherplatzkomplexität einer Konfiguration gleich der längsten Beschriftung eines Arbeitsbandes.

Definition 5.1.4. Die **Speicherplatzkomplexität von** M ist eine Funktion $\text{Space}_M : \mathbb{N} \to \mathbb{N}$ definiert durch

$$\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid x \in \Sigma^n\}$$

Lemma 5.1.1. *Sei $k \in \mathbb{N} - \{0\}$ und A eine k-Band-TM, die immer hält. Dann existiert eine äquivalente 1-Band-TM B, so dass*

$$\text{Space}_B(n) \leq \text{Space}_A(n)$$

Lemma 5.1.2. *Sei $k \in \mathbb{N} - \{0\}$ und A eine k-Band-Turingmaschine. Für jede solche MTM existiert eine k-Band-TM B, so dass $L(A) = L(B)$ und*

$$\text{Space}_B(n) \leq \frac{\text{Space}_A(n)}{2} + 2.$$

Definition 5.1.5. Für jede Funktion $f : \mathbb{N} \to \mathbb{R}^+$ sei

$$\mathcal{O}(f(n)) = \{r : \mathbb{N} \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{N} : \forall n \geq n_0 : r(n) \leq c \cdot f(n)\}.$$

Jede Funktion $r \in \mathcal{O}(f(n))$ wächst **asymptotisch nicht schneller** als f.

Definition 5.1.6. Für jede Funktion $g : \mathbb{N} \to \mathbb{R}^+$ sei

$$\Omega(g(n)) = \{s : \mathbb{N} \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}, \exists d \in \mathbb{N} : \forall n \geq n_0 : s(n) \geq \frac{1}{d} \cdot g(n)\}.$$

Für jede Funktion $s \in \Omega(g(n))$ sagen wir, dass die Funktion s **asymptotisch mindestens so schnell wächst wie** g.

Definition 5.1.7. Für jede Funktion $h : \mathbb{N} \to \mathbb{R}^+$ sei

$$\Theta(h(n)) = \mathcal{O}(h(n)) \cap \Omega(h(n)).$$

Falls $g \in \Theta(h(n))$ so sagen wir, dass g und h **asymptotisch gleich sind**.

Definition 5.1.8. Seien $f, g : \mathbb{N} \to \mathbb{R}^+$. Falls

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0,$$

dann **wächst g asymptotisch schneller als f** und wir schreiben $f(n) = o(g(n))$.

Definition 5.1.9. Sei L eine Sprache und $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion.

- $\mathcal{O}(f(n))$ ist eine **obere Schranke für die Zeitkomplexität**, wenn es eine MTM A gibt, mit $L(A) = L$ und $\text{Time}_A(n) \in \mathcal{O}(f(n))$.

- $\Omega(f(n))$ ist eine **untere Schranke für die Zeitkomplexität**, wenn es eine MTM A gibt, mit $L(A) = L$ und $\text{Time}_A(n) \in \Omega(f(n))$.
- Eine MTM A heisst **optimal für L**, falls $\text{Time}_C(n) \in \mathcal{O}(f(n)) \cap \Omega(f(n))$.

5.2 Komplexitätsklassen

Die Komplexitätsklassen sind Sprachklassen. Wir betragen also Mengen von Entscheidungsproblemen.

Definition 5.2.1. Für alle Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ seien:

- $\text{TIME}(f) = \{L(M) \mid M \text{ ist eine MTM}, \text{Time}_M(n) \in \mathcal{O}(f(n))\}$
- $\text{SPACE}(g) = \{L(M) \mid M \text{ ist eine MTM}, \text{Space}_M(n) \in \mathcal{O}(g(n))\}$
- $\text{DLOG} = \text{SPACE}(\log_2(n))$
- $\text{P} = \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c)$
- $\text{PSPACE} = \bigcup_{c \in \mathbb{N}} \text{SPACE}(n^c)$
- $\text{EXPTIME} = \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{n^d})$

Lemma 5.2.1. *$\forall t : \mathbb{N} \to \mathbb{R}^+$ gilt* $\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$.

Korollar 5.2.1. $\text{P} \subseteq \text{PSPACE}$

Definition 5.2.2. Eine Funktion $s : \mathbb{N} \to \mathbb{N}$ heisst **platzkonstruierbar**, falls eine 1-Band-TM M existiert, so dass gilt:

- $\forall n \in \mathbb{N} : \text{Space}_M(n) \leq s(n)$
- für jede Eingabe O^n und $n \in \mathbb{N}$ generiert M das Wort $0^{s(n)}$ auf seinem Arbeitsband und hält in q_{accept}.

Definition 5.2.3. Ein Funktion $t : \mathbb{N} \to \mathbb{N}$ heisst **zeitkonstruierbar**, falls eine MTM A existiert, so dass gilt:

- $\text{Time}_A(n) \in \mathcal{O}(t(n))$
- für jede Eingabe $0^n, n \in \mathbb{N}$ generiert A das Wort $0^{t(n)}$ auf dem ersten Arbeitsband und hält in q_{accept}

Lemma 5.2.2. *Sei $s : \mathbb{N} \to \mathbb{N}$ eine platzkonstruierbare Funktion. Sei M eine MTM mit $\forall x \in L(M) : \text{Space}_M(x) \leq s(|x|)$. Dann existiert eine MTM A mit $L(A) = L(M)$ und $\text{Space}_A(n) \leq s(n)$. Daraus folgt direkt $\forall y \in \Sigma_A^* : \text{Space}_A(y) \leq s(|y|)$.*

Dieses Lemma zeigt, dass es für jede platzkonstruierbare Funktion ausreicht, einen MTM M mit $L(M) = L$ mit $s(n)$-platzbeschränkten Berechnungen auf allen Eingaben aus L zu konstruieren, um die Existenz einer MTM A zu garantieren,

die ebenfalls L akzeptiert und auf allen Eingaben (auch solchen aus L^C die Schranke $s(n)$ für die Platzkomplexität einhält.

Analog zu diesem Lemma, gibt es auch eines für die Zeitkomplexität.

Lemma 5.2.3. *Sei $t : \mathbb{N} \to \mathbb{N}$ eine zeitkonstruierbare Funktion. Sei M eine MTM mit $\forall x \in L(M) : \text{Time}_M(x) \leq t(|x|)$. Dann existiert eine MTM A mit $L(A) = L(M)$ und $\text{Time}_A(n) \in (O)(t(n))$.*

Satz 5.2.1. Für jede platzkonstruierbare Funktion s mit $s(n) \geq \log_2(n)$ gilt $\text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$.

Korollar 5.2.2. DLOG $\subseteq$ P *und* PSPACE $\subseteq$ EXPTIME

5.3 Nichtdeterministische Komplexitätsmasse

Definition 5.3.1. Sei M eine MTM oder eine nichtdeterministische MTM und $x \in L(M) \subseteq \Sigma^*$. Die **Zeitkomplexität von M auf** x, $\text{Time}_M(x)$, ist die Länge der kürzesten akzeptierenden Berechnung von M auf x. Die **Zeitkomplexität von** M ist die Funktion $\text{Time}_M : \mathbb{N} \to \mathbb{N}$, definiert durch

$$\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid x \in L(M) \cap \Sigma^n\}$$

Definition 5.3.2. Sei $C = C_1, C_2, \ldots, C_m$ eine akzeptierende Berechnung von M auf x. Sei $\text{Space}_M(C_i)$ die Speicherkomplexiät der Konfiguration C_i. Dann wird folgendes definiert:

$$\text{Space}_M(C) = \max\{\text{Space}_M(C_i) \mid i = 1, 2, \ldots, m\}$$

Definition 5.3.3. Die **Speicherplatzkomplexität von M auf** x ist dann

$\text{Space}_M(x) = \min\{\text{Space}_M(C) \mid C$ ist eine akzeptierende Berechnung von M auf x$\}$.

Definition 5.3.4. Die **Speicherplatzkomplexität von** M ist die Funktion $\text{Space}_M : \mathbb{N} \to \mathbb{N}$ definiert durch

$$\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid x \in L(M) \cap \Sigma^n\}.$$

Definition 5.3.5. Für alle Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ sei:

- $\text{NTIME}(f) = \{L(M) \mid M$ eine nichtdeterministische MTM mit $\text{Time}_M(n) \in \mathcal{O}(f(n))\}$

- $\text{NSPACE}(g) = \{L(M) \mid M \text{ ist eine nichtdeterministische MTM mit } \text{Time}_M(n) \in \mathcal{O}(g(n))\}$
- $\text{NLOG} = \text{NSPACE}(\log_2(n))$
- $\text{NP} = \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c)$
- $\text{NPSPACE} = \bigcup_{c \in \mathbb{N}} \text{NPSPACE}(n^c)$

Lemma 5.3.1.
- $\text{NTIME}(t) \subseteq \text{NSPACE}(t)$
- $\text{NSPACE}(t) \subseteq \bigcup_{c \in \mathbb{N}} \text{NTIME}(c^{s(n)})$

Satz 5.3.1. Für jede Funktion $t : \mathbb{N} \to \mathbb{R}^+$ und jede zeit- und platzkonstruierbare Funktion $s : \mathbb{N} \to \mathbb{N}$ mit $s(b) \geq \log_2(b)$ gilt:

- $\text{TIME}(t) \subseteq \text{NTIME}(t)$
- $\text{SPACE}(t) \subseteq \text{NSPACE}(t)$
- $\text{NTIME}(s(n)) \subseteq \text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$

Korollar 5.3.1.

- $\text{NP} \subseteq \text{PSPACE}$
- $\text{NLOG} \subseteq \text{P}$
- $\text{NPSPACE} \subseteq \text{EXPTIME}$
- $\text{PSPACE} = \text{NPSPACE}$

Daraus erhalten wir die **fundamentale Komplexitätsklassenhierarchie der sequentiellen Berechnungen**

$$\text{DLOG} \subseteq \text{NLOG} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

5.4 NP und Beweisverifikation

Definition 5.4.1.

$$\text{SAT} = \{x \in \Sigma^*_{\text{logic}} | x \text{ kodiert eine erfüllbare Formel in KNF}\}$$

Definition 5.4.2. Sei $L \subseteq \Sigma^*$ eine Sprache und $p : \mathbb{N} \to \mathbb{N}$ eine Funktion. Eine MTM A ist ein **p-Verifizierer für L**, $V(A) = L$, falls A mit folgenden Eigenschaften auf allen Eingaben aus $\Sigma^* \times \Sigma^*_{\text{bool}}$ arbeitet:

1. $\text{Time}_A(w, x) \leq p(|w|)$
2. Für jedes $w \in L$ existiert ein $x \in \Sigma^*_{\text{bool}}$, so dass $|x| \leq p(|w|)$ und $(w, x) \in L(A)$. Das Wort x nennt man dabei den **Beweis** oder **Zeugen** der Behauptung $w \in L$.
3. Für jedes $y \notin L$ gilt $(y, z) \notin L(A)$ für alle $z \in \Sigma^*_{\text{bool}}$

Definition 5.4.3. Die MTM A ist ein **Polynomialzeit-Verifizierer**, falls A ein Verifizierer ist und $p(n) \in \mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$.

Definition 5.4.4. Die Klasse der in **Polynomialzeit verifizierbaren Sprachen** ist

$$\text{VP} = \{V(A) \mid A \text{ ist ein Polynomialzeit-Verifizierer}\}$$

Bemerkung 5.4.1. Für einen p-Verifizierer A sind $L(A)$ und $V(A)$ unterschiedliche Sprachen:

$$V(A) = \{w \in \Sigma^* \mid \exists x \in \Sigma^*_{\text{bool}} : |x| \leq p(|w|) \wedge (w, x) \in L(A)\}$$

Somit ist A ein deterministischer Algorithmus, der für eine Eingabe (w, x) verifiziert, ob x ein Beweis für "$w \in L$" ist. A verifiziert erfolgreich, wenn $w \in V(A)$ und somit ein Beweis x für "$w \in L$" gibt mit $|x| \leq p(|w|)$. Die Gleichheit $V(A) = L$ fordert $\forall w \in L : \exists x \in \Sigma^*_{\text{bool}} : |x| \leq p(|w|)$.

Satz 5.4.1. $VP = NP$

5.5 NP-Vollständigkeit

Das Konzept der NP-Vollständigkeit basiert auf der Annahme, dass $P \subsetneq NP$!

Definition 5.5.1. $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ zwei Sprachen. Dann ist L_1 **polynomiell auf** L_2 **reduzierbar**, $L_1 \leq_p L_2$, falls eine polynomielle TM A existiert, so dass für jedes Wort $x \in \Sigma_1^*$ ein Wort $A(x) \in \Sigma_2^*$ erzeugt wird und gilt:

$$x \in L_1 \Leftrightarrow A(x) \in L_2.$$

In einem solchen Fall wird A als **polynomielle Reduktion** bezeichnet.

Somit entspricht die polynomielle Reduktion der EE-Reduktion, nur mit der zusätzlichen Forderung, dass die dabei erzeugte TM A in polynomieller Zeit läuft.

Definition 5.5.2. Eine Sprache L ist **NP-schwer**, falls gilt:

$$\forall L' \in NP : L' \leq_p L.$$

Definition 5.5.3. Eine Sprache L ist **NP-vollständig**, falls

- $L \in NP$
- L ist NP-schwer

Lemma 5.5.1. *Falls $L \in P$ und L ist NP-schwer, dann muss $P = NP$ gelten.*

Satz 5.5.1. SAT ist NP-vollständig.

Lemma 5.5.2. *Seien L_1, L_2 zwei Sprachen. Falls $L_1 \leq_p L_2$ und L_1 ist NP-schwer, dann ist auch L_2 NP-schwer.*

Sprachen Definitionen:

- SAT = $\{\Phi \mid \Phi$ ist eine erfüllbare Formel in KNF$\}$
- 3SAT = $\{\Phi \mid \Phi$ ist eine erfüllbare Formel in 3KNF$\}$
- CLIQUE = $\{(G,k) \mid G$ ist ein ungerichteter Graph, der eine k-Clique enthält$\}$
- VC = $\{(G,k) \mid G$ ist ein ungerichteter Graph und Knotenüberdeckung der Mächtigkeit höchstens $k\}$

Für eine Formel Φ sei φ eine Belegung der Variabeln von Φ. Somit bezeichnet $\varphi(\Phi)$ den Wahrheitswert von Φ bei der Belegung φ.

Lemma 5.5.3. SAT $\leq_p$ CLIQUE

Proof. Sei $\Phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$ eine Formel in KNF. Somit gilt $F_i = (l_{i,1} \vee l_{i,2} \vee \ldots \vee l_{i,k_i})$ für $k_i \in \mathbb{N} - \{0\}$ und $i = 1, 2, \ldots, m$.

Es soll nun aus der Eingabe einer KNF Formel Φ die Eingabe G, k für das Cliquenproblem erstellt werden, so dass

$$\Phi \in \text{SAT} \Leftrightarrow (G,k) \in \text{CLIQUE}\,.$$

Dies geschieht durch folgende Zuweisungen:

- $k = m$
- $G = (V, E)$ wobei:
  - $V = \{[i,j] \mid 1 \leq i \leq m \wedge 1 \leq j \leq k_i\}$. Wir erzeugen also für jedes Literal in Φ einen Knoten in G.
  - $E = \{\{[i,j],[r,s]\} \mid \forall [i,j],[r,s] \in V \wedge i \neq r \wedge l_{i,j} \neq \overline{l_{r,s}}\}$. Somit verbindet eine Kante $\{[i,j],[r,s]\}$ nur Knoten aus unterschiedlichen Klauseln (die F_i) und achtet darauf, dass das Literal von $[i,j]$ nicht die Negation des Literals von $[r,s]$ ist.

Es ist somit klar, dass wir mittels eines polynomiellen Algorithmus aus Φ das Cliqueproblem (G,k) generieren können. Es bleibt nun zu zeigen, dass

$$\Phi \text{ ist erfüllbar } \Leftrightarrow G \text{ enthält eine Clique der Grösse } k = m.$$

Grundidee für den Beweis: Zwei Literale $l_{i,j}$ und $l_{r,s}$ sind in G verbunden, wenn beide Literale aus unterschiedlichen Klauseln stammen und beide gleichzeitig den Wert 1 annehmen können. Somit entspricht eine Clique in G der Belegung von Variabeln von Φ, die die Literale der Knoten der Clique erfüllen (zu 1 auswerten).

Der Beweis wird in beide Richtungen geführt.

"$\Rightarrow$": Sei Φ eine erfüllbare Formel. Dann existiert eine Belegung φ, so dass $\varphi(\Phi) = 1$. Daraus folgt, dass für alle Klauseln F_i zu gelten hat, dass $\varphi(F_i) = 1$.

Daraus wiederrum folgt, dass in einer Klausel F_i für mindestens ein Literal l_{i,α_i} zu gelten hat, dass $\varphi(l_{i,\alpha_i}) = 1$ wobei $\alpha_i \in \{1, \ldots, k_i\}$.

Es ist klar, dass $[1, \alpha_1], [2, \alpha_2], \ldots, [m, \alpha_m]$ aus unterschiedlichen Klauseln stammen.

Die Gleichheit $l_{i,\alpha_i} = \overline{l_{j,\alpha_j}}$ für beliebige $i \neq j$ impliziert, dass für jede Belegung ω gelten würde: $\omega(l_{i,\alpha_i}) = \omega(l_{j,\alpha_j}$ und daraus wiederum folgt, dass $\varphi(l_{i,\alpha_i}) = \varphi(l_{j,\alpha_j}) = 1$ nicht möglich wäre. Also ist $l_{i,\alpha_i} \neq \overline{l_{j,\alpha_j}}$ für alle $i \neq j$ und $\{[i, \alpha_i], [j, \alpha_j]\} \in E$. Somit ist $\{[i, \alpha_i] \mid 1 \leq i \leq m\}$ eine Clique der Grösse m.

"$\Leftarrow$": Sei Q eine Clique von G mit $k = m$ Knoten. Wir wissen, dass zwei Knoten durch eine Kante in G nur dann verbunden sind, wenn sie zwei Literale aus unterschiedlichen Klauseln entsprechen. Es existieren somit $\alpha_1, \alpha_2, \ldots, \alpha_m, \alpha_p \in \{1, 2, \ldots, k_p\}$ für $p = 1, \ldots, m$, so dass $\{[1, \alpha_1], [2, \alpha_2], \ldots, [m, \alpha_m]\}$ die Knoten von Q sind. Somit gibt es eine Belegung φ der Variablen von Φ, so dass $\varphi(l_{1,\alpha_1}) = \varphi(l_{2,\alpha_2}) = \cdots = \varphi(l_{m,\alpha_m}) = 1$. Dies wiederum impliziert direkt, dass gilt: $\varphi(F_1) = \varphi(F_2) = \cdots = \varphi(F_m) = 1$ und so erfüllt die Belegung φ die Formel Φ.

□

Lemma 5.5.4. CLIQUE $\leq_p$ VC

Lemma 5.5.5. SAT $\leq_p$ 3SAT

Definition 5.5.4. NPO ist die Klasse der Optimierungsprobleme, wobei

$$U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal}) \in NPO,$$

falls folgende Bedingungen erfüllt sind:

- $L \in P$. Es kann also effizient überprüft werden, ob ein $x \in \Sigma_I^*$ eine zulässige Eingabe für U ist.
- es existiert ein Polynom p_U, so dass
  - für jedes $x \in L$ und jedes $y \in \mathcal{M}(x) : |y| \leq p_U(|x|)$. Jede zulässige Lösung von U ist polynomiell in der Eingabegrösse.
  - es existiert ein polynomieller Algorithmus A, der für jedes $y \in \Sigma_O^*$ und jedes $x \in L$ mit $|y| \leq p_U(|x|)$ entscheidet, ob $y \in \mathcal{M}(x)$ oder nicht.
- die Funktion cost kann man in polynomieller Zeit berechnen.

Ein Optimierungsproblem U ist also in NPO, falls

- es möglich ist effizient zu überprüfen, ob ein Wort im Problemfall liegt. Also ob ein Wort eine Instanz von U ist,
- die Grösse der Lösung polynomiell in der Grösse der Eingabe ist und man in polynomieller Zeit verifizieren kann, ob ein y eine gültige Lösung für einen gegebenen Problemfall ist und

- die Kosten der zulässigen Lösung effizient berechnen kann.

Definition 5.5.5. PO ist die Klasse von Optimierungsproblemen $U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, so dass

- $U \in NPO$
- es existiert ein polynomieller Algorithmus A, so dass für jedes $x \in L$, $A(x)$ eine optimale Lösung für x ist.

Definition 5.5.6. Sei $U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ ein Optimierungsproblem aus NPO ($U \in NPO$). Die **Schwellenwert-Sprache für** U ist

$$\text{Lang}_U = \{(x, a) \in L \times \Sigma^*_{\text{bool}} \mid \text{Opt}_U \leq \text{Nummer}(a)\}$$

falls goal = Minimum, und für goal = Maximum:

$$\text{Lang}_U = \{(x, a) \in L \times \Sigma^*_{\text{bool}} \mid \text{Opt}_U \geq \text{Nummer}(a)\}.$$

U ist **NP-schwer**, falls Lang_U NP-schwer ist.

Lemma 5.5.6. *Falls das Optimierungsproblem $U \in PO$, dann ist* $\text{Lang}_U \in P$.

Lemma 5.5.7.

- *MAX-SAT ist NP-schwer*
- *MAX-CL ist NP-schwer*

Chapter 6

Grammatiken und Chomsky-Hierarchie

Buchkapitel 10.1 bis und mit 10.3 sind Teil des ersten Selbststudiums

6.1 Einleitung

Endliche Automaten (EA) und Turingmaschinen (TM) erlauben es unendliche Objekte wie Sprachen und Mengen in endlicher Form zu beschreiben. Grammatiken sind eine weitere Möglichkeit Sprachen in endlicher und eindeutiger Weise zu beschreiben.

6.2 Grammatiken

Beispiel 10.1?

Definition 6.2.1. Eine Grammatik G ist ein 4-Tupel $G = (\Sigma_N, \Sigma_T, P, S)$ wobei die Bedeutung folgende ist:

- Das Alphabet Σ_N ist das **Nichtterminalalphabet**. Die Symbole aus Σ_N nennt man **Nichtterminale**.
- Das Alphabet Σ_T ist das **Terminalalphabet**. Die Symbole aus Σ_T nennt man **Terminalsymbole**.
- $S \in \Sigma_N$ und ist das **Startsymbol**. Somit muss die Generierung eines Wortes jeweils mit dem Wort $w = S$ beginnen.
- P ist eine endliche Teilmenge von $\Sigma^*\Sigma_N\Sigma^* \times \Sigma^*$ wobei $\Sigma = \Sigma_N \cup \Sigma_T$ ist. Die Elemente von P heissen **Regeln**. Statt $(\alpha, \beta) \in P$ zu schreiben, wird meist $\alpha \rightarrow_G \beta$ geschrieben. Wobei die Bedeutung wie folgt ist: α kann in G durch β ersetzt werden.

Bemerkung 6.2.1. In dieser Vorlesung gilt:

- Kleinbuchstaben a, b, c, d, e und Ziffern werden für Terminalsymbole verwendet.
- Grossbuchstaben A, B, C, D, X, Y, Z werden für Nichtterminale verwendet.
- Mit Kleinbuchstaben u, v, w, x, y, z werden Wörter über Σ_T bezeichnet.
- Mit griechischen Kleinbuchstaben $(\alpha, \beta, \gamma, \ldots)$ werden beliebige Wörter über $\Sigma = \Sigma_T \cup \Sigma_N$ bezeichnet.

Bemerkung 6.2.2. Folgende Dinge sind zu beachten:

- Es hat jeweils $\Sigma_N \cap \Sigma_T = \emptyset$ zu gelten.
- Durch die Anforderung $\alpha \in \Sigma^* \Sigma_N \Sigma^*$ muss α mindestens ein Nichtterminal enthalten.

Definition 6.2.2. Sei $\gamma, \delta \in (\Sigma_N \cup \Sigma_T)^*$. δ ist aus γ in einem Ableitungsschritt in G ableitbar, $\gamma \Rightarrow_G \delta$, genau dann, wenn $\omega_1, \omega_2 \in (\Sigma_N \cup \Sigma_T)^*$ und eine Regel $(\alpha, \beta) \in P$ existieren, so dass gilt: $\gamma = \omega_1 \alpha \omega_2$ und $\delta = \omega_1 \beta \omega_2$.

δ ist aus γ ableitbar in G, $\gamma \Rightarrow_G^* \delta$, genau dann wenn

- entweder $\gamma = \delta$,
- oder für ein $n \in \mathbb{N} - \{0\}$ und $n+1$ Wörter $\omega_1, \omega_2, \ldots, \omega_n \in (\Sigma_N \cup \Sigma_T)^*$ existieren, so dass $\gamma = \omega_0, \delta = \omega_n$ und $\omega_i \Rightarrow_G \omega_{i+1}$ für $i = 0, 1, 2, \ldots, n-1$.

In anderen Worten: Falls $\gamma \Rightarrow_G^* \delta$ gilt, so gibt es eine Folge von Ableitungsschritten, die bei $\gamma = \omega_1$ beginnt und bei $\delta = \omega_n$ endet.

Somit ist $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$.

Definition 6.2.3. Falls $\omega \in \Sigma_T^*$ und $S \Rightarrow_G^* \omega$ gilt, dann sagt man, dass ω von G erzeugt wird. Die von G erzeugte Sprache ist somit $L(G) = \{\omega \in \Sigma_T^* | S \Rightarrow_G^* \omega\}$.

Bemerkung 6.2.3. Sei $\alpha \Rightarrow_G^i \beta$ eine Ableitung im Sinne von $\alpha \Rightarrow_G^* \beta$, die aus genau i Schritten besteht.

Grammatiken sind nichtdeterministische Erzeugungsmechanismen für Sprachen. Dies kommt daher, dass es mehrere gleiche linke Seiten geben darf und die Wahl der Anwendung einer dieser nicht festgelegt ist.

Beweistypen einbauen

6.3 Chomsky-Hierarchie

Definition 6.3.1. Sei $G = (\Sigma_N, \Sigma_T, P, S)$ eine Grammatik

- G ist eine **Typ-0-Grammatik**. Die Typ-0-Grammatik ist die Klasse aller uneingeschränkten Grammatiken.
- G ist **kontextsensitiv** oder **Typ-1-Grammatik**, falls $\forall(\alpha, \beta) \in P : |\alpha| \leq |\beta|$ gilt. Es gibt somit nicht die Möglichkeit ein Teilwort α durch ein kürzeres Teilwort β zu ersetzen.
- G ist **kontextfrei** oder **Typ-2-Grammatik**, falls $\forall(\alpha, \beta) \in P : \alpha \in \Sigma_N \wedge \beta \in (\Sigma_N \cup \Sigma_T)^*$ gilt. Alle Regeln haben also die Grundform $X \to \beta$ für ein Nichtterminal $X \in \Sigma_N$.
- G ist **regulär** oder **Typ-3-Grammatik**, falls $\forall(\alpha, \beta) \in P : \alpha \in \Sigma_N \wedge \beta \in (\Sigma_T^* \cdot \Sigma_N \cup \Sigma_T^*)$. Somit haben alle Regeln einer regulären Grammatik entweder die Form $X \to u$ oder $X \to uY$ für $u \in \Sigma_T^*$ und $X, Y \in \Sigma_N$. Auch ist anzumerken, dass das Nichtterminal immer ganz rechts auf der rechten Seite zu stehen hat.

Bemerkung 6.3.1. Eine Sprache ist vom Typ i ($i = 0, 1, 2, 3$), falls sie durch eine Typ-i-Grammatik erzeugt werden kann.

Kontextfreie Sprachen haben die Eigenschaft, dass Nichtterminale unabhängig von den benachbarten Symbolen ersetzt werden können.

6.4 Reguläre Grammatiken und endliche Automaten

Lemma 6.4.1. *$\mathcal{L}_3$ enthält alle endlichen Sprachen.*

Lemma 6.4.2. *$\mathcal{L}_3$ ist abgeschlossen bezüglich der Vereinigung. Somit gilt für alle Sprachen $L_1, L_2 \in \mathcal{L}_3 : L_1 \cup L_2 \in \mathcal{L}_3$.*

Lemma 6.4.3. *$\mathcal{L}_3$ ist abgeschlossen bezüglich der Konkatenation. Somit gilt für alle Sprachen $L_1, L_2 \in \mathcal{L}_3 : L_1 \cdot L_2 \in \mathcal{L}_3$.*

Satz 6.4.1. Zu jedem endlichen Automaten (EA) A existiert eine reguläre Grammatik G mit $L(A) = L(G)$.

Definition 6.4.1. Eine reguläre Grammatik (Typ-3-Grammatik) $G = (\Sigma_N, \Sigma_T, P, S)$ heisst **normiert**, wenn alle Regeln der Grammatik nur eine der folgenden drei Formen haben:

- $S \to \lambda$, wobei S das Startsymbol ist
- $A \to a$, wobei $A \in \Sigma_N$ und $a \in \Sigma_T$
- $B \to bC$, wobei $B, C \in \Sigma_N$ und $b \in \Sigma_T$

Lemma 6.4.4. *Für jede reguläre Grammatik G existiert eine äquivalente **normierte** reguläre Grammatik G'.*

Eine nicht normierte reguläre Grammatik G kann dabei wie folgt in eine äquivalente und normierte reguläre Grammatik G' überführt werden:

- **Kettenregeln:** Regeln der Form $X \to Y$, $X, Y \in \Sigma_N$ enden nach endlich vielen Ableitungsschritten in $\alpha \in (\Sigma_T^* \cup \Sigma_T^+ \cdot \Sigma_N)$. Somit ersetzen wir $X \to Y$ durch $X \to \alpha$.
- Alle Regeln der Form $A \to \lambda$ für $A \in \Sigma_N - \{S\}$. In einer normierten regulären Grammatik darf nur aus dem Startsymbol das leere Wort abgeleitet werden. Dazu betrachten wir die Regeln $B \to \omega A$, $A \to \lambda$ womit durch das hinzufügen der Regel $B \to \omega$ und entfernen der Regel $A \to \lambda$ das Problem gelöst wird.

Satz 6.4.2. $\mathcal{L}_3 = \mathcal{L}(EA)$

6.5 Kontextfreie Grammatiken und Kellerautomaten

Bemerkung 6.5.1.

- Die Klasse $\mathcal{L}_2 = \mathcal{L}_{CF}$. Die Klasse der kontextfreien Sprachen ist also gleich der Klasse der Typ-2-Grammatiken.
- $\mathcal{L}_3 \subset \mathcal{L}_{CF}$. Somit sind alle regulären Sprachen auch kontextfreie Sprachen bzw. es gibt kontextfreie Sprachen, die keine regulären Sprachen sind.

Definition 6.5.1. Sei $G = (\Sigma_N, \Sigma_T, P, S)$ eine kontextfreie Grammatik und sei das Wort $x \in L(G)$. Sei a eine Ableitung von x in G. Der Syntaxbaum (Ableitungsbaum) T_a der Ableitung a ist ein markierter Baum mit folgenden Eigenschaften:

- T_a hat eine Wurzel, die mit S markiert ist.
- Jeder Knoten von T_a ist mit einem Symbol aus $\Sigma_N \cup \Sigma_T \cup \{\lambda\}$ markiert.
- Innere Knoten sind mit Symbolen aus Σ_N markiert.
- Alle Blätter sind mit Symbolen aus $Sigma_T \cup \{\lambda\}$ markiert. Gelesen von links nach rechts ergeben sie das Wort x.
- Für den ersten Ableitungsschritt $S \to \beta_1\beta_2\beta_3 \dots \beta_r$, wobei $\beta_i \in \Sigma_N \cup \Sigma_T$, erhält S genau r Söhne $v_1, \dots, v_r$, die von links nach rechts mit $\beta_1, \dots, \beta_r$ markiert sind.
- Für jede Anwendung einer Regel der Form $A \to \alpha_1\alpha 2 \dots \alpha_k$ in a hat der entsprechende innere Knoten v mit der Markierung A genau k Söhne $v_1, \dots, v_k$, die von links nach rechts mit den Symbolen $\alpha_1, \dots, \alpha_k$ markiert sind. Wobei $\alpha_i \in \Sigma_N \cup \Sigma_T$.

Der Syntaxbaum T_a wird auch als Syntaxbaum zur Generierung von x in G bezeichnet.

Jede Ableitung hat einen eindeutigen Syntaxbaum. Wobei mehrere unterschiedliche Ableitungen zu dem gleichen Syntaxbaum führen können.

Definition 6.5.2. Eine kontextfreie Grammatik G ist **normiert**, wenn sie

- keine nutzlosen Symbole enthält. Also alle Symbole X, die startend von S nicht erreicht werden können.
- keine Regel der Form $X \to \lambda$ hat.
- keine Kettenregeln bezitzt.

Definition 6.5.3. Sei $G = (\Sigma_N, \Sigma_T, P, S)$ eine kontextfreie Grammatik. G ist in **Chomsky-Normalform**, falls alle Regeln eine der beiden Formen haben:

- $A \to BC$ für $A, B, C \in \Sigma_N$, oder
- $A \to a$ für $A \in \Sigma_N, a \in \Sigma_T$.

Definition 6.5.4. G ist in **Greibach-Normalform**, wenn alle Regeln die folgende Form haben:

$$A \to a\alpha \quad \text{mit } A \in \Sigma_N, a \in \Sigma_T, \alpha \in \Sigma_N^*.$$

Satz 6.5.1. Für jede kontextfreie Grammatik G mit $\lambda \notin L(G)$ existiert eine zu G äquivalente Grammatik in Chomsky-Normalform und in Greibach-Normalform.

Lemma 6.5.1 (Pumping-Lemma für kontextfreie Sprachen)**.** *Für jede kontextfreie Sprache L gibt es eine nur von L abhängige Konstante n_L, so dass für alle Wörter $z \in L$ mit $|z| \geq n_L$ eine Zerlegung $z = uvwxy$ existiert, so dass*

- $|vx| \geq 1$,
- $|vwx| \leq n_L$, *und*
- $\{uv^iwx^iy \mid i \in \mathbb{N}\} \subseteq L\}$.

Seite 386

Chapter 7

Reguläre Ausdrücke

Thema des 2. Selbststudiums. Basierend auf dem Buch *Introduction to Automata Theory, Languages, and Computation* von *John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman*.

7.1 Endliche Automaten mit Epsilon-Übergängen

Nichtdeterministische Endliche Automaten (NEA) werden nun um Übergänge auf dem leeren Wort (ϵ) erweitert. Ist aus einem Zustand ein solcher Übergang zu einem anderen Zustand vorhanden, so kann der NEA sich "spontan" entscheiden diesen auszuführen, ohne eine Eingabe zu lesen.

Diese Erweiterung des NEA macht ihn aber im Bezug auf die Sprachklasse, die durch einen "normalen" NEA akzeptiert werden, nicht mächtiger. Nachfolgend werden solche erweiterten NEA als **ϵ-NEA** bezeichnet.

Wichtig ist, dass das Symbol für das leere Wort nicht Teil des zugrundeliegenden Alphabets ist.

Ein ϵ-NEA entspricht in der formalen Definition einem NEA, bis auf die Erweiterung der Übergangsfunktion δ, welche nun definiert ist als

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q).$$

7.1.1 Epsilon-Abschluss

Informell geht es beim Epsilon-Abschluss darum die Menge aller Zustände zu finden, die von einem Startzustand erreichbar sind, in dem man nur Pfaden folgt, die mit einem ϵ gekennzeichnet sind. Man möchte also alle Zustände haben, die von einem Startzustand erreichbar sind, ohne ein Symbol in der Eingabe gelesen zu haben.

Definition 7.1.1. Der **Epsilon-Abschluss**, ECLOSE(q), eines Zustands $q \in Q$ ist rekursiv wie folgt definiert:

Beginn: Der Startzustand q ist in ECLOSE(q).

Rekursion: Wenn ein Zustand $p \in$ ECLOSE(q) und es gibt einen Übergang von p zum Zustand r, der mit ϵ beschriftet ist, dann ist gilt auch $r \in$ ECLOSE(q). Formal ausgedrückt:

$$p \in \text{ECLOSE}(q) \Rightarrow \delta(p, \epsilon) \in \text{ECLOSE}(q)$$

7.2 Reguläre Ausdrücke

Definition 7.2.1. Für einen regulären Ausdruck E ist $L(E)$ die Sprache, die durch den regulären Ausdruck beschrieben wird.

Definition 7.2.2. Reguläre Ausdrücke werden rekursiv definiert:

Beginn:

- Die Konstanten ϵ und $\emptyset$ sind reguläre Ausdrucke. Dabei gilt $L(\epsilon) = \{\epsilon\}$ und $L(\emptyset) = \emptyset$.
- Ein Symbol $a \in \Sigma$ ist ein regulärer Ausdruck. Dabei gilt $L(a) = \{a\}$.

Induktion:

- Wenn E, F zwei reguläre Ausdrücke sind, dann ist auch $E + F$ ein regulärer Ausdruck. Dabei beschreibt $E + F$ die Vereinigung zweier Sprachen: $L(E + F) = L(E) \cup L(F)$.
- Wenn E, F zwei reguläre Ausdrücke sind, dann ist auch EF ein regulärer Ausdruck. Dabei beschreibt EF die Konkatenation zweier Sprachen: $L(EF) = L(E) \cdot L(F)$.
- Wenn E ein regulärer Ausdruck ist, dann ist auch E^* ein regulärer Ausdruck. Dabei gilt: $L(E^*) = (L(E))^*$.
- Wenn E ein regulärer Ausdruck ist, dann ist auch (E) ein regulärer Ausdruck. Dabei gilt: $L((E)) = L(E)$.

7.2.1 Reguläre Ausdrücke und endliche Automaten

7.2.1.1 EA zu regulärem Ausdruck

Sei der EA A gegeben und wir möchten nun den regulären Ausdruck R konstruieren, so dass $L(A) = L(R)$.

Sei $Q = \{1, 2, \ldots, n\}$ die Zustände von A. Dies können wir durch umbenennen der Zustände von A erreichen im allgemeinen Fall.

Nun definieren wir $R_{i,j}^{(k)}$ als den regulären Ausdruck, dessen Sprache ($L(R)$) eine Menge von Wörtern w ist, so dass jedes $w \in L(R)$ einen Pfad beschreibt im EA A vom Zustand $i \in Q$ zum Zustand $j \in Q$ und gleichzeitig keinen inneren

Knoten $p \in Q$ enthält mit $p \geq k$. Dabei ist ein innerer Knoten jeder Knoten im Pfad, der nicht gleich i oder j ist.

Um $R_{i,j}^{(k)}$ zu konstruieren, wird folgender induktiver Ansatz gewählt. Dieser beginnt bei $k = 0$ und endet mit $k = n$.

Beginn: Sei $k = 0$. Da alle Zustände 1 oder höher sind, ist die hier vorhandene Restriktion, dass der Pfad keine inneren Knoten hat. Es gibt genau zwei Arten von Pfaden mit dieser Eigenschaft:

- Ein Übergang vom Zustand i direkt zum Zustand j.
- Ein Pfad der Länge 0, der nur aus einem Zustand i besteht.

Wenn $i \neq j$, dann kann nur der erste Fall zutreffen. Somit suchen wir im EA A nach Eingabesymbolen $a \in \Sigma$, die zu einem Übergang von i zu j auslösen würden.

1. Falls kein solches a existiert, dann gilt $R_{i,j}^{(0)} = \emptyset$.
2. Falls genau ein solches a existiert, dann gilt $R_{i,j}^{(0)} = a$.
3. Falls es mehrere solche Symbole gibt, also $a_1, a_2, \ldots, a_k$, dann gilt $R_{i,j}^{(0)} = a_1 + a_2 + \cdots a_k$.

Wenn aber $i = j$, dann sind nur Pfade der Länge 0 erlaubt und alle Schleifen von i zu sich selbst. Solche Pfade der Länge 0 werden im regulären Ausdruck durch ϵ beschrieben. Somit fügen wir ϵ mittels Vereinigung zu den vorhin erzeugten regulären ausdrücken. Es gilt also:

1. Falls kein solches a existiert, dann gilt $R_{i,j}^{(0)} = \epsilon$.
2. Falls genau ein solches a existiert, dann gilt $R_{i,j}^{(0)} = \epsilon + a$.
3. Falls es mehrere solche Symbole gibt, also $a_1, a_2, \ldots, a_k$, dann gilt $R_{i,j}^{(0)} = \epsilon + a_1 + a_2 + \cdots a_k$.

Induktion: Gehen wir davon aus, dass es einen Pfad von i nach j gibt, der durch keinen Zustand mit höherem Wert als k geht. Dann gibt es zwei Möglichkeiten:

1. Der Pfad geht nie durch den Zustand k. In diesem Fall gilt $R_{i,j}^{(k)} = R_{i,j}^{(k-1)}$.
2. Der Pfad geht mindestens einmal durch den Zustand k. Dann lässt sich dieser Pfad in mehrere Teile aufteilen. Das erste Stück geht von i zu k ohne k als inneren Knoten zu haben und das letzte Stück geht von k zu j, ohne k als inneren Knoten zu haben. Somit können wir folgenden regulären Ausdruck verwenden:

$$R_{i,j}^{(k)} = \underbrace{R_{i,k}^{(k-1)}}_{\text{erstes Stück}} \underbrace{(R_{k,k}^{(k-1)})^*}_{\substack{\text{mittlere Stücke, die beliebig von} \\ k \text{ zu } k \text{ verlaufen}}} \underbrace{R_{k,j}^{(k-1)}}_{\text{letztes Stück}}$$

Kombinieren wir beide Möglichkeiten, so erhalten wir

$$R_{i,j}^{(k)} = R_{i,j}^{(k-1)} + R_{i,k}^{(k-1)}(R_{k,k}^{(k-1)})^* R_{k,j}^{(k-1)}$$

Es reicht also, wenn wir beim kleinsten k beginnen und Schritt für Schritt diesen erhöhen. So gelangen wir schlussendlich zum $R_{1,j}^{(n)}$, wobei $j \in F$, also ein akzeptierender Zustand.

Regulären Ausdruck bauen mittels entfernen von Zuständen ausgelassen, da sehr visuell ist

(N)EA bauen aus einem regulären Ausdruck ausgelassen, da sehr visuell ist

Chapter 8

Hilfreiches und Verschiedenes

8.1 Hamiltonischer Kreis

Ein **Hamiltonischer Kreis** eines Graphen G ist ein geschlossener Weg, der jeden Knoten von G genau einmal enthält.

8.2 Traveling Salesman Problem (TSP)

Fragestellung: Gegeben ist eine Menge von Städten und der Reisedistanz zwischen diesen Städten. Gesucht ist die kürzeste Route mit welcher alle Städte genau einmal besucht werden und mit der man wieder beim Anfang landet.

Es handelt sich um ein NP-vollständiges Problem.

Zur Lösung dieses Problems wird der kürzeste Hamiltonsche Kreis gesucht, der am gegebenen Startpunkt beginnt.

8.3 Minimum Vertex Cover Problem (MIN-VCP)

Gesucht wird die minimale Knotenmenge (**??**), die eine Knotenüberdeckung eines Graphen ist.

8.4 Maximaler Schnitt (MAX-CUT)

Seite 257

8.5 MAX-SAT

Seite 256

8.6 Sprachklassen

8.6.1 Klassenübersicht

Übersicht und Bedeutung

8.6.2 Sprachenzugehörigkeit

Übersicht welche Sprachen zu welcher Klasse gehören

8.7 Arithmetische Operationen auf MTMs

Sei hier jeweils M eine MTM. Zu dieser MTM gibt es immer eine äquivalente 1-Band-TM (die Definition von Platzkonstruierbarkeit verlangt zum Beispiel eine 1-Band-TM). Stehe auf dem ersten Arbeitsband 0^n und auf dem zweiten Arbeitsband 1^m. Das dritte Arbeitsband soll $0^n \square m$ enthalten, wobei $\square \in \{+, -, \times, \div, \ldots\}$.

8.7.1 Addition

Auf dem dritten Arbeitsband soll 0^{n+m} stehen.

Dazu liest M das erste Arbeitsband und schreibt auf das dritte eine 0 für jede gelesene 0. Hat M die letzte 0 gelesen, so beginnt M mit dem einlesen des zweiten Arbeitsbandes. Für jede gelesene 1 schreibt M eine 0 auf das dritte Arbeitsband. Danach geht M in q_{accept} über.

Auf dem dritten Arbeitsband steht nun 0^{n+m}.

8.7.2 Subtraktion

Auf dem dritten Arbeitsband soll 0^{n-m} stehen, wobei wir hier von $n \geq m$ ausgehen.

Dazu schreibt M eine 0 auf das dritte Arbeitsband für jede gelesene 0 auf dem ersten Arbeitsband. Danach liest M das zweite Arbeitsband und löscht von rechts nach links je eine 0 für jede gelesene 1 auf dem zweiten Arbeitsband. Danach geht M in q_{accept} über.

Auf dem dritten Arbeitsband steht nun 0^{n-m}.

Falls $n < m$, so muss eine zusätzliche Abbruchbedingung hinzugefügt werden.

8.7.3 Multiplikation

Auf dem dritten Arbeitsband soll $0^{n \times m}$ stehen.

M liest schrittweise die 0-en auf dem ersten Arbeitsband. Für jede gelesene 0 liest M alle 1-en vom zweiten Arbeitsband und für jede gelesene 1 schreibt es eine 0 auf das dritte Arbeitsband. Nachdem alle 1 gelesen sind, fährt M den Lese-/Schreibkopf an den Anfang vom zweiten Arbeitsband, bevor es das nächste Zeichen auf dem ersten Arbeitsband verarbeitet. Sind alle 0-en auf dem ersten Arbeitsband gelesen, so geht M in q_{accept} über.

Auf dem dritten Arbeitsband steht nun $0^{n \times m}$.

8.7.4 Division

Auf dem dritten Arbeitsband soll $0^{\lfloor n \div m \rfloor}$ stehen.

Dazu liest M für jede 1 auf dem zweiten Arbeitsband eine 0 auf dem ersten Arbeitsband. Ist M am Ende des zweiten Arbeitsbandes, so schreibt M eine 0 auf das dritte Arbeitsband. M fährt den Lese-/Schreibkopf auf dem zweiten Arbeitsband zurück und wiederholt das vorgehen, bis keine 0 mehr gelesen werden kann auf dem ersten Arbeitsband. In diesem Fall geht M in q_{accept} über.

Auf dem dritten Arbeitsband steht nun $0^{\lfloor n \div m \rfloor}$.

Falls $0^{\lceil n \div m \rceil}$ gesucht ist, so schreibt man immer zuerst die 0 auf das dritte Arbeitsband und liest dann so viele 0-en auf dem ersten Band, wie 1-en auf dem zweiten Arbeitsband.

8.7.5 Logarithmus zur Basis 2

Auf dem dritten Arbeitsband soll $0^{\lceil \log_2(n+1) \rceil}$ stehen.

M liest das zweite Eingabeband. Für jede gelesene 0 schreibt M die Position auf das dritte Arbeitsbands. Dies macht sie wie folgt: M initialisiert das dritte Arbeitsband mit einer 0. Für jede gelesene 0 vom zweiten Arbeitsband addiert M eine 1 zum Wert, der auf dem dritten Arbeitsband kodiert ist. Hat M alle 0-en auf dem zweiten Eingabeband gelesen, so ist auf dem dritten Arbeitsband die Länge des Wortes binär kodiert. Dies entspricht gerade $\lceil \log_2(n+1) \rceil = \text{Bin}(n)$.

Somit steht auf dem dritten Arbeitsband nun $0^{\lceil \log_2(n+1) \rceil}$.

8.8 Formeln

8.8.1 Summenformel für Anzahl Programme

$$\sum_{i=k}^{n} 2^i = 2^{n+1} - 2^k$$

Chapter 9

Vorlesung

9.1 Vorlesung vom 17.09.2013

- Zwei Prüfungen unter dem Semester: Mitte Semester und zweite Woche vor Semesterende. Sessionsprüfung muss man so oder so gehen. Durchschnitt der Semesterprüfungen sind die minimale Abschlussnote. Sessionsprüfung kann diese Note nur verbessern. Für Teilnahme an der Semesterprüfungen muss man den Grossteil der Serien gelöst haben
- Einführung in die Geschichte der Mathematik und die Entstehung der Informatik. Was unterscheidet die Informatik von anderen Wissenschaften?
- Alphabet, Wort und Wortlänge eingeführt
- Buchthemen übersprungen: Kodierung von Zahlen, Graphen und anderen Dingen als Wort
- Konkatenation eingeführt

9.2 Vorlesung vom 20.09.2013

- $(\Sigma^*, \cdot)$ ist ein Monoid
- Teilwörter eingeführt
- Beispiel: gegeben ein Wort der Länge k, wie viele unterschiedliche Teilwörter gibt es?
- $w \in \Sigma^*, |w|_a$ eingeführt (Anzahl Vorkommen von a in w)
- Kanonische Ordnung nochmals kurz repetiert
- Sprachen eingeführt
- Potenznotation bei Buchstaben eingeführt: $x^2 = xx = x \cdot x$
- Konkatenation von Sprachen eingeführt

- Potenzennotation bei Sprachen eingeführt
- Kleensche Stern eingeführt
- Mengenoperationen auf Sprachen
  - Beweis für $L_1L_2 \cup L_1L_3 = L_1(L_2 \cup L_3)$
  - $L_1L_2 \cap L_1L_3 \neq L_1(L_2 \cap L_3)$ erklärt
- Homomorphismus eingeführt
- Mit Einführung in Algorithmische Probleme begonnen

9.3 Vorlesung vom 24.09.2013

- Entscheidungsproblem
- Algorithmus um ein Wort $w \in \Sigma^*$ auszugeben ohne Eingabe (Aufzählungsalgorithmus)
- offene Frage: Wie misst man den Informationsgehalt in Texten/Wörtern
- Shannon Entropie angesprochen (wahrscheinlich als Exkurs)
- Komprimierung/Kodierung zur Messung des Informationsgehalts angesprochen, gezeigt wieso es sich nicht wirklich anbietet
- Kolmogorov-Komplexität eingeführt

9.4 Vorlesung vom 27.09.2013

- Was passiert mit der Kolmogorov-Komplexität, wenn eine andere Programmiersprache verwendet wird?
- Definition des "Zufalls" durch Kolmogorov-Komplexität. Es geht dabei nicht darum, wie wahrscheinlich/zufällig ein Ereignis ist (Wahrscheinlichkeit), sondern wie zufällig ein Objekt ist, dass wir vor uns haben. Ein Objekt ist dabei zufällig, wenn man es nur durch sich selbst beschrieben werden kann und nicht durch eine kürzere Form.
- Primzahlensatz mit Kolmogorov-Komplexität